D1.1 – Applied gaming asset methodology

Realising an Applied Gaming Ecosystem

Research and Innovation Action

Grant agreement no.: 644187

D1.1 – Applied gaming asset methodology

RAGE – WP1 – D1.1

Project Number	H2020-ICT-2014-1
Due Date	M29: 30 June 2017
Actual Date	30 June 2017
Document Author/s	Westera, W., Stefanov, K., Van der Vegt, W., Nyamsuren, E., Bahreini, K., Kluijfhout, E., Moreno Ger, P., Freire, M., Georgiev, A., Grigorov, A., Boytchev, P., Griffiths, D., Fernández Magnón, B., Mascarenhas, S., Martinez Ortiz, I., & Hemmje, M.
Version	1.0
Dissemination level	PU
Status	FINAL
Document approved by	Wim

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644187

Document Version Control			
Version	**Date**	**Change Made (and if appropriate reason for change)**	**Initials of Commentator(s) or Author(s)**
0.1	May 2017	Initial version D1.1 draft	WimW
0.2	May 23, 2017	D1.1 version 0.2	WimW, IvanM, EKL, WimV, PB, SamM
0.3	May 31, 2017	Usability study added RAGE viewer added WP1/6 coordination document included	ENY, KBA, PB WimW, MH, KS
1.0	June 30, 2017	Review comments processed	WimW

Document Change Commentator or Author		
Author Initials	**Name of Author**	**Institution**
WimW	Wim Westera	1.OUNL
IvanM	Iván Martínez-Ortiz	2. UCM
EKL	Eric Kluijfhout	1. OUNL
WimV	Wim van der Vegt	1. OUNL
PB	Pavel Boytchev	15. SU
SamM	Samuel Mascarenhas	3. INESC ID
ENY	Enkhbold Nyamsuren	1. OUNL
KBA	Kiavash Bahreini	1. OUNL
MH	Matthias Hemmje	5. FTK
KS	Krassen Stefanov	15. SU

Document Quality Control			
Version QA	**Date**	**Comments (and if appropriate reason for change)**	**Initials of QA Person**
0.3	June 11	More details needed, e.g. usability study details	M. Kickmeier-Rust

0.3	June 24	Style issues Various changes in annex 2, 3 and 4	J. Jeuring

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

EXECUTIVE SUMMARY

Together with the intermediate deliverable D1.4 (Month 18), which was prepared for the first project review, this document explains how the RAGE project defines, develops, distributes and maintains a series of applied gaming software assets that it aims to make available. It describes a high-level methodology and infrastructure that are needed to support the work in the project as well as after the project has ended. As to avoid unnecessary duplication the contents of D1.4 are presented as a summary. In the current deliverable (D1.1) the asset creation methodology, the quality assurance considerations and the asset metadata requirements are merged together and implemented into a single asset creation wizard, which supports and guides asset owners through the process of asset submission to the Ecosystem portal. It complements the metadata editor that was developed earlier, but which in some respects turned out to be demanding for asset developers. The wizard design, which was based on an analysis of the asset submission workflow, decomposes the submission process into 8 subsequent steps, while a limited subset of metadata fields are qualified as mandatory. The wizard was used and evaluated by all RAGE's asset developers. Also, the metadata-viewer tool is briefly explained in this deliverable.
Herewith the methodology and the upfront tools for creating assets are ready for wider use. In the period after completion of this deliverable, the asset creation part (the asset repository and authoring tools) described here will be part of the overall RAGE ecosystem portal. For the alignment of the two subsystems a coordination document was jointly created by WP1 and WP6, and has been included here. Already before the (soft) external launch of the ecosystem portal, which is scheduled in month 36 (January 2018) external parties will be involved to explore the asset creation system and make judgements about its usability. Additional fine-tuning of the wizard and its instructions is anticipated.
Overall, the asset creation part and its alignment with the RAGE ecosystem portal has now been fully covered.

1 INTRODUCTION

Purpose of this document
This document describes a high-level methodology and infrastructure that are needed to create, upload and maintain a set of software assets for applied gaming. This deliverable D1.1 was preceded by the intermediate version D1.4 (Month 18), which was prepared for the first project review. As to avoid any unnecessary duplication the contents of D1.4 are presented as a summary below.

1.1 Main outcomes reported in intermediate deliverable D1.4

The main results described in intermediate deliverable D1.4 (M18) are presented below in condensed form. For details we refer to D1.4 and its appendices.

Analysis of technical landscape
An in-depth analysis of the technical landscape of game engines, platforms and programming languages was presented and used to restrict asset development to a few primary code bases that still would allow to reach out to a maximum number of game developers and their platforms. RAGE will particularly focus on C# and TypeScript (typed JavaScript).

Asset architecture
An asset system architecture was designed to support both server-side assets and client-side assets. Platform and hardware dependencies were avoided as much as possible as to achieve maximum portability between game engines, programming languages and platforms. Server-side assets will provide client-side companion assets or at least will provide REST web services. The client-side asset architecture was validated for multiple platforms and languages (C#, TypeScript as well as JavaScript, Java, C++).

Interoperability
Interoperability between assets is covered by RAGE's component-based asset architecture, With respect to interoperability standards and specifications we have adopted a case-based approach, focussing on asset interoperability in the RAGE pilots. Interoperability with external systems focuses on the application of xAPI.

Asset metadata schema
An asset metadata XML schema was designed to accommodate search in the asset repository, and to include dependencies, software versions, ownership and licensing information. The schema is based on a core subset of RAS and extends it with elements from ADMS, IEEE LOM and metadata related to the applied games domain.

Asset development methodology
The RAGE asset development methodology was presented, while it assumes neutrality towards different software development environments, programming languages and methodologies. Starting points for asset quality assurance have been specified for further elaboration and implementation.

Asset licensing
After detailed analysis the Apache version 2.0 license has been proposed as the default license to be granted to RAGE software assets, because it offers protection and openness, and it allows for commercial exploitation. The proposal has been officially adopted and confirmed by the RAGE Strategic Board.

1.2 Recent work

Complementing the work presented in Deliverable 1.4, current deliverable (D1.1) presents:

- The asset metadata editor pre-evaluation
  A small-scale evaluation of the metadata editor was carried out
- The asset creation wizard

A workflow guidance wizard was developed to facilitate metadata entry and the uploading of artefacts

- The asset creation wizard usability test and data entry
  The wizard was tested by all RAGE asset developers, who all were asked to enter their asset's metadata and artefacts
- The asset metadata viewer
- Coordination of asset creation and asset publication
  This covers the alignment of asset repository and asset creation tools with the ecosystem portal, which is the publishing and community environment

In accordance with the work plan, completion of these results was achieved before month 30 as to allow internal launch and testing of the RAGE ecosystem platform by month 32, and the soft external launch scheduled by the end of year 3 (month 35).

2 RAGE METADATA EDITOR PRE-EVALUATION

The RAGE metadata editor is a front-end tool for the asset repository, that allows asset developers to enter and edit the metadata associated with an asset and any of its artefacts. The editor hides the internal metadata complexity and constructs a flexible dynamic interface. Figure 1 shows a screenshot of the editor.

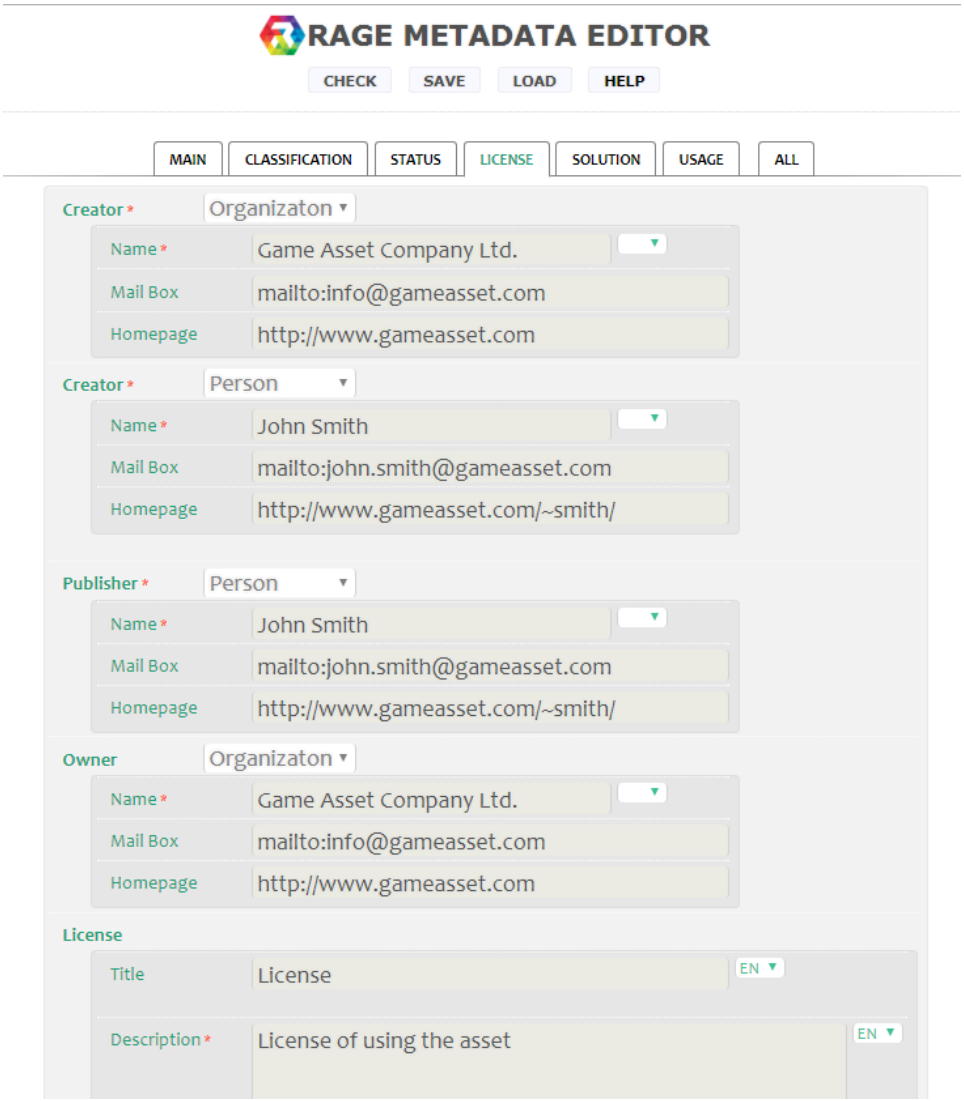


Figure 1. Screenshot example of the metadata editor.

The RAGE Metadata editor is actually a meta-editor. This is, it is not an editor by itself, but it builds a metadata editor in real time – see Figure 2. The main, central element is the RAGE Metadata Meta-editor. It contains a collection of building blocks and construction algorithms. The inputs to the editor are heterogeneous definitions of metadata. This includes a metadata model, additional schemas, taxonomies and styling preferences. Then the meta-editor constructs the façade (the graphical user interface) of an editor (as shown in Figure 4), which is presented to the user. The constructed metadata editor is targeted towards a very specific metadata set – the one defined in the input data.

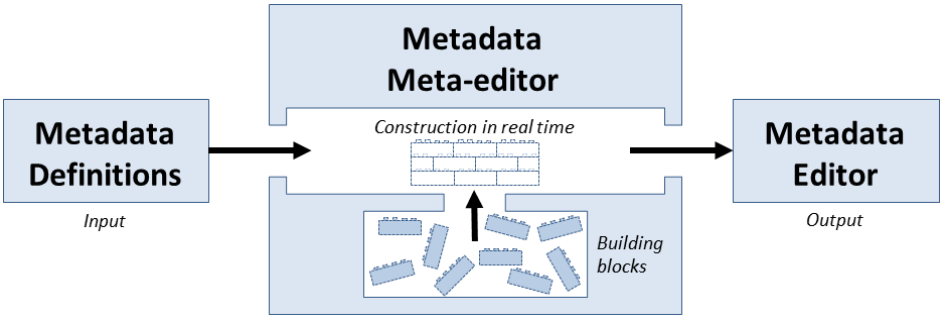


Figure 2. Metadata meta-editor

The technical details of the metadata editor as well as a technical pre-evaluation were reported in a conference paper about the RAGE infrastructure (see annex 1, http://dspace.ou.nl/handle/1820/7329):

Georgiev, A., Grigorov, A., Bontchev, B., Boytchev, P., Stefanov, K., Westera, W., Prada, E., Hollins, P. and Moreno Ger, P. (2016). The RAGE Advanced Game Technologies Repository for Supporting Applied Game Development. In R. Bottina, J. Jeuring and M. Veltkamp (Eds.), Proceedings of the 5th International Conference, GALA 2016, Utrecht, The Netherlands, December 5-7, 2016, (pp. 235-245). Cham: Springer International Publishing. doi=10.1007/978-3-319-50182-6_21.

An extended version of this conference paper was invited for submission to the International Journal of Serious Games (status unknown at the time of writing).

In the study, four usage scenarios were tested:

1. Publishing/updating a game asset
   The asset developer signs in, creates/selects an asset, enters/updates metadata and uploads artefacts or an asset package.
2. Publishing/updating a game asset from GitHub
   The asset developer signs in, creates/selects an asset, provides the GitHub repository identifier and credentials (if required). We expect to be able to realise automated harvesting of respective files (artefacts) and metadata from GitHub in the future (using the GitHub API).
3. Publishing/updating a game asset from an IDE.
   For this scenario we developed a proof of concept for the Eclipse IDE plugin. The asset developer opens the asset project in the Eclipse IDE; using the plugin the developer creates/updates the asset in RAGE Asset Repository within the IDE, providing credentials and needed metadata.
4. Search for assets
   This scenario involves full text or advanced search, browsing the repository, viewing assets' metadata and downloading assets or artefacts for reuse.

Nine end-users, viz. asset developers from RAGE, were involved in the tests. Results showed that the technical operations of the editor are valid. Also, users can easily work with basic services such as searching, downloading or uploading assets to the repository. Nevertheless, for some of the metadata fields users would need more specific instructions about how to populate the repository with metadata. Although the overall conclusion is that RAGE end-users accept the editor as a usable tool for entering their metadata, provided that more detailed documentation would become available, we took into consideration that external technology developers who want to upload their components to the RAGE repository, might be deterred by the complexity of the metadata and its documentation, and –worst case- withdraw. In order to arrive at a sustainable ecosystem with a continuous influx of new technology assets from external parties, the metadata barrier should be as low as possible. Therefore, we have decided to design and develop a workflow guidance wizard on top of the editor, which facilitates a

stepwise process of metadata entry, without the need to read extensive documentation. This wizard is presented and explained in the next chapter.

3 ASSET CREATION WIZARD

Based on the outcomes of the pre-evaluation study of the metadata editor (cf. chapter 2) we have decided to develop asset creation wizard that guides asset developers through the process of metadata and artefacts entry. Before describing the wizard we first explain our starting points.

3.1 Starting points

Starting points for the asset quality assurance approach
The following quality assurance starting points have been identified. The RAGE asset quality assurance methodology should (cf. Deliverable D1.4):

- be lightweight, in order not to turn off external asset providers
- require minimal effort from asset developers to complete
- not duplicate existing systems and tools
- expose a minimum set of essential requirements
- be neutral with respect to different software development environments
- be neutral towards different asset programming methodologies
- be neutral with respect to different programming languages
- be neutral with respect to different game development platforms
- cover the workflow across the asset lifecycle
- require minimal involvement of QA Staff to minimise exploitation costs
- be sustainable beyond the ending of the project

Starting points for the asset creation wizard

- The wizard should guide the asset developer in a few steps through the metadata and artefacts entry process.
- It should be effective and usable.
- Progress indicators should be included to keep asset developers informed and motivated.
- A limited set of mandatory metadata elements should be identified.
- A limited set of optional metadata elements should be included.
- The systems should allow for benefitting from external communities and versioning systems (e.g. Github, Bitbucket).
- Possibly include automated checks and balances.
- As not all data can be checked, the wizard goes with a (legal) self-declaration by the end-user about the quality and correctness of the metadata provided.

Altogether, these requirements imply that on many occasions a trade-off will be needed between simplicity of quality assurance and completeness. Still, given the fact that assets are developed with the goal of being integrated in third-party development projects, the RAGE Quality Assurance approach should cover aspects beyond standard practices for source code, with a strong emphasis on reliable metadata, documentation, demonstrators and supporting materials.

3.2 Software maturity

With respect to software quality it is important to note that early software releases (e.g. alpha versions, beta versions) should be welcomed, as it would be in agreement with the common practices in open software communities. This implies that full compliance with RAGE software quality standards can only be demanded for version 1.0 or higher. Below, we will first explain the software versioning model that will be used as a maturity indicator. Subsequently, we will go into potential software quality indicators.
The maturity of software is often communicated by developers through the version numbers. Requiring developers to indicate the maturity of their code in other ways may lead to unnecessary duplication of information, and to demotivating developers to provide that

information. Therefore we make use of software version numbers to communicate maturity information. In order for this to be effective, a standard system for the numbering of releases needs to be adopted across RAGE assets. The system adopted is the Semantic Versioning Specification 2.0.0 (SemVer) available at http://semver.org/.

The specification summarises itself as:

Given a version number MAJOR.MINOR.PATCH, increment the:

> *MAJOR version when you make incompatible API changes,*
> *MINOR version when you add functionality in a backwards-compatible manner, and*
> *PATCH version when you make backwards-compatible bug fixes.*

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

Two particularly relevant points for the purposes of RAGE are items 4 and 5 of the specification, which distinguish between initial development and a public offering:

- *4. Major version zero (0.y.z) is for initial development. Anything may change at any time. The public API should not be considered stable.*
- *5. Version 1.0.0 defines the public API. The way in which the version number is incremented after this release is dependent on this public API and how it changes.*

3.3 Minimal subset of mandatory metadata

High quality of asset metadata is paramount for findability by end-users. For lowering the proverbial burden of entering metadata, the smallest possible set of mandatory metadata field was defined. Some optional fields are also presented in the wizard. Table 1 provides the overview of mandatory (and optional) metadata fields covered by the wizard.

Table 1. Prioritised RAGE metadata schema elements

No	Metadata	Explanation	Mandatory	Optional
1	Name	Asset title	X	
2	One sentence description	Text describing the purpose of the asset	X	
3	Short non-technical description	Text describing non-technical details and used for short advertisements	X	
4	Technical description	Text describing technical details of the asset	X	
5	Picture/Logo	An image to represent the software asset	X	
6	Date	Relevant date, for instance the date of release		X
7	Language	The language of software messages		X
8	Access URL	The public address of the software asset. It may point to the asset's home page, or its GitHub address.		X
9	Game development environment	The game environment that asset was (initially) created for.		X
10	Target platform	The target platform for which the asset was built, e.g. Android, Windows, IOS		X
11	Programming	The programming language used		X

	language	for creating the asset		
12	Applied computing concepts	Keywords from the ACM computing Classification System	X	
13	Learning goals	Any specific learning goals that the asset might address		X
14	Keywords	Open field for terms and concepts that help to characterise the asset		X
15	Version	SVS code assigned by the asset developer	X	
16	Version notes	Details about this version	X	
17	Development status	Can either be “under development”, “completed”, “depricated”, or “withdrawn”	X	
18	Commit URL	Reference to an external versioning system		X
19	License type	E.g. Apache 2.0, GPL2	X	
20	License URL	Link to the online license version		X
21	Conditions and restrictions	Textual explanation of any constraints.	X	
22	Owner	Mandatory name and optional contact details (home page and email address) of the software asset owner	X	
23	Creators	References to one or more organisations or individuals who created the software (names, home pages, email addresses)	X	
24	Detailed description	Extended, technical description explaining what the asset does, what inputs it needs, how it functions, technical requirements, operational constraints, etc.		X
25	Source code	Various files or links of the software	X	
26	Documentation	Various files or links	X	
27	Setup files	Files with e.g. installation scripts, setup guides	X	
28	Test	Files with test suites and documentation		X
29	Other resources	E.g. design documents, data files, examples		X
30	Coding style	Characterisation of coding, e.g. code validation, style guides used	X	
31	Architecture	Compliance with RAGE architecture	X	
32	Software testing	Performed tests, such as unit tests, integration, performance.		X
33	Self-declaration for correctness		X	
34	Self-declaration for responsibility		X	

3.4 Wizard description

Based on the outcomes of the pre-evaluation study of the metadata editor (cf. chapter 2) we have decided to develop a wizard to guide asset developers through the entire process of

metadata entry. The wizard decomposes the process into 8 successive steps along the most relevant parts of the RAGE metadata scheme (which is kept hidden). The 8 steps of the wizard are:

1. About (metadata fields 1 to 8 from **Fout! Verwijzingsbron niet gevonden.**)
   Requiring general information, e.g. title, description
2. Classification (metadata fields 9 to 14 from **Fout! Verwijzingsbron niet gevonden.**)
   Requiring info about target platforms, programming language, applied computing keywords)
3. Status (metadata fields 15 to 18 from **Fout! Verwijzingsbron niet gevonden.**)
   Info about the software version, version notes, commit reference
4. License (metadata fields 19 to 21 from **Fout! Verwijzingsbron niet gevonden.**)
   Details about the license(s), conditions and potential restrictions
5. Contacts (metadata fields 22 to 23 from **Fout! Verwijzingsbron niet gevonden.**)
   Information about owners and creators
6. Resources (metadata fields 24 to 29 from **Fout! Verwijzingsbron niet gevonden.**)
   Files or references of the software, documentation, tests, etc.
7. Quality (metadata fields 30 to 34 from **Fout! Verwijzingsbron niet gevonden.**)
   Information about the asset's quality
8. Submission

A screenshot of step 1 of the wizard is displayed in Figure 3.

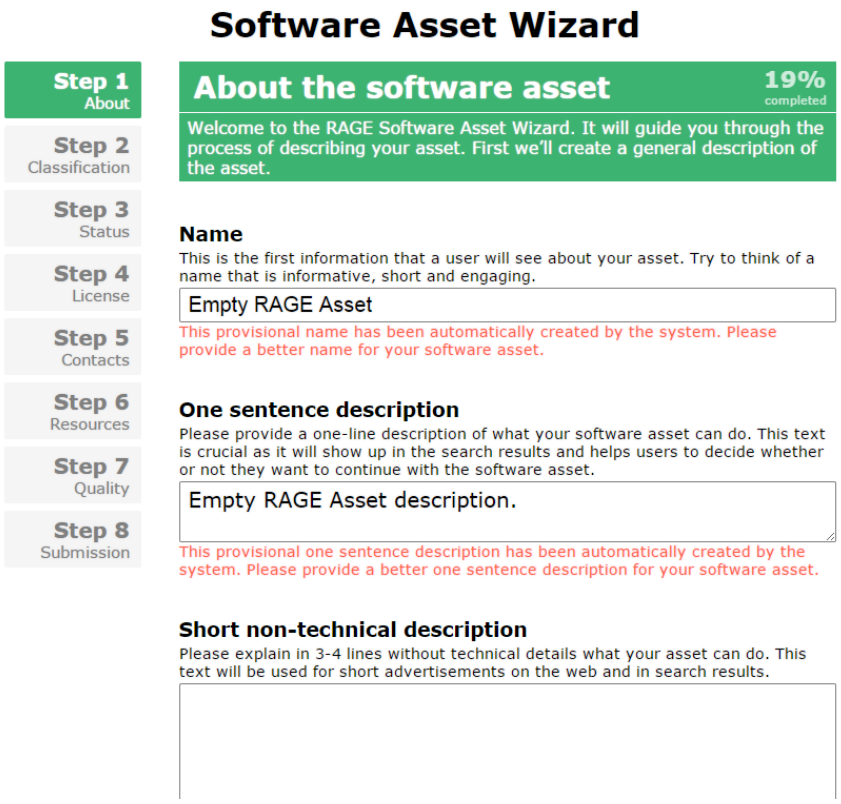


Figure 3. Screenshot of the asset creation wizard (version May 21st).

In each step various mandatory as well as optional metadata fields are displayed. The vertical menu on the left of the screen provides an overview of the steps and allows for easy navigation between steps. To inform end-users about their progression to keep them motivated, weighted metrics based on the mandatory fields are used to display the completion rate of the process (cf. "19%" in Figure 3). Also, asset quality is estimated on the basis of the data entered in step 7

with respect to compliance with the RAGE architecture, the declared coding style and the testing efforts made (fields 30-32). Going through the 8 steps of the wizard would be enough to capture all relevant metadata in most cases. Still, the original metadata editor remains accessible also.

Manuals
In addition to the on-screen guidance, the wizard also refers to detailed manuals. These include:

- A manual how to comply with the client-side RAGE asset architecture in C# (Annex 2)
- A manual how to comply with the client-side RAGE asset architecture in TypeScript (Annex 3)
- A manual for doing a code-review check for client-side assets (Annex 4).

Legal self-declaration
As not all data can be automatically checked by the system, the approach relies on a self-declaration clause that is approved by the user before submission. The user declares that all entered information is correct and that he or she takes sole responsibility for all content posted and activity that occurs under the account.

3.5 The price of simplicity

The simplicity of the wizard and its self-declaration approach may be beneficial for both asset developers and ecosystem managers, but the potential downside cannot remain undiscussed.
As only limited checks and balances can be used to assess the quality and correctness of the submitted software, metadata and artefacts, the approach strongly relies on the integrity of the submitters. Claims made by asset developers may be unjust, brushed up, or subject to mistakes. The self-declaration required for each submission makes clear that the asset developer is the sole responsible and liable person. Based on common practice in existing IT communities we suppose that community feedback such as public comments and ratings from asset consumers, will generate sufficient self-cleaning power to signal, remove and prevent low quality assets.
By allowing premature asset versions, the RAGE portal may expose more diverse, advanced and experimental solutions, be it at the expense of software completeness and robustness. By conforming to the Semantic Versioning Specification, RAGE assures that early version software is clearly earmarked and distinguishable from mature versions.
Asset developers are free to decide whether their software files are stored in the RAGE repository or on an external platform. This is because RAGE does not want to duplicate external software versioning and software management systems, such as Github. Also, this allows software developers to join RAGE without giving up their preferred software management platforms. At the downside RAGE becomes dependent on external platforms: if the external service closes down, cf. Google Code and Microsoft´s CodePlex, relevant software may be lost. The same holds for other externally stored artefacts, such as videos, slides, and documents. Nevertheless, RAGE assumes that software owners will then be able to preserve their software in time.
Finally, quality problems may occur in the asset's installation guides and user manuals as well as in instructional videos, slides and other media artefacts, without being checked by RAGE: the RAGE platform just accepts these artefacts as files, without any quality procedures. By exposing good quality manual and materials for its own assets, RAGE aims to implicitly set the quality standards for these. We are considering the option of presenting "Featured Assets", which are guaranteed to have been thoroughly reviewed through a special application procedure that involves further analysis. In the end, the community members will decide what to appreciate and what to reject.

4 USABILITY STUDY OF THE ASSET CREATION WIZARD

4.1 Measurement instruments

We have used three validated measurement instruments for evaluating the usability of the Asset Creation Wizard, namely, System Usability Scale (SUS) [1], Form Usability Scale (FUS) [2,3], and the Usability Metric for User Experience (UMUX) [4].
SUS is a well-validated and reliable questionnaire applicable to a wide range of software systems. However, the questions in the SUS questionnaire are too generic to evaluate usability issues specific to online forms. For this reason, we have also used the FUS questionnaire, which was specifically designed for evaluating the usability of online forms. While the SUS and the FUS questionnaires provide overall scores of usability, the UMUX questionnaire was developed to explicitly reflect the four separate components of usability as defined by ISO 9241-11: efficiency, effectiveness, satisfaction, and overall usability.
An optional open input field to comment the answer followed each question in the measurement instrument.

System Usability Scale. The SUS questionnaire consists of 10 questions. Responses are measured on a Likert scale ranging from 1 (strongly disagree) to 5 (strongly agree). The score of each odd question is the scale position minus one, while the score of each even question is five minus the scale position. The sum of all the scores is multiplied by 2.5 to obtain the overall score per participant ranging from 0 to 100 with 50 as being neutral.

Form Usability Scale. The FUS questionnaire consists of 10 questions. Validation of the questionnaire showed that question 7 provides little discriminatory value and information gain [2,3]. Therefore, the question is excluded from our analysis.
Responses are measured on a Likert scale ranging from 1 (strongly disagree) to 6 (strongly agree). Additionally, there is an option to skip each question. The overall score per participant is obtained by computing the mean of all questions.
In our study, we used a 5-point Likert scale and removed the option for skipping. We normalized the FUS score to make it comparable with the SUS score. For each question, the score was calculated as the scale position minus one. The sum of all scores is divided by 36 and multiplied by 100 to obtain the overall score ranging from 0 to 100.

Usability Metric for User Experience. The UMUX questionnaire consists of four questions. The two questions regarding the satisfaction and overall usability overlap with two questions from the SUS questionnaire. Therefore, we have reused responses for the SUS questionnaire for evaluating these two usability components.
Originally, all questions are measured on a Likert scale between 1 (strongly disagree) to 7 (strongly agree). However, we used a 5-point Likert scale for the purpose of consistency across questionnaires. Consecutively, odd questions are scored as the scale position minus one and even questions are scored as five minus the scale position. The overall score is obtained by dividing the sum of four scores by 16 and then multiplying by 100.

4.2 Results

The user data was gathered between May 5th, 2017 and May 29th, 2017, over the internet. In total, 15 asset developers participated in the study. The asset developers were members of the RAGE project from eight different institutes. They were instructed to use the wizard for submitting the metadata and artefacts of their game assets to the RAGE repository. All participants managed to successfully complete their submissions. Participants were asked to address and complete the questionnaire(s) after usage of the wizard.
This sections summarises the main outcomes of the usability study. For more details the reader is referred to Annex 5.

SUS results. The mean overall SUS score is 72.8 (*SD*=16.3, *SE*=4.2) where *SD* and *SE* are the standard deviation and the standard error, respectively. This overall usability evaluation is positive. All the participants except one positively evaluated the overall usability of the Wizard. The mean scores for all questions except one are positive. The question with the negative mean

score (M=1.7, SE=.3) is concerned with the frequency of using the Wizard. The negative score is expected because the Wizard is supposed not to be frequently used.

FUS results. The mean overall FUS score is 65.7 (*SD*=16.0, *SE*=4.1). Overall usability evaluation of the Wizard is positive with some rooms for improvement. Two participants negatively evaluated the overall usability of the Wizard. The correlation between the SUS and the FUS overall scores is significantly high (r(13) = .67, p = .006), which indicates that the FUS score is consistent with the benchmark score of the SUS.
The overall FUS score is lower than the overall SUS score. The lower score might indicate that the FUS questionnaire is able to identify issues specific to online forms. To further investigate this matter, we look at the scores of the individual FUS questions. The mean scores for nine questions are positive. One question has a negative mean score (Mean=1.9, SE=.3). The responses to this question indicate that the Wizard did not have sufficient feedback to users for resolving unexpected problems. Another question with the lowest mean positive score (M=2.3, SE=.3) is close to the neutral score of 2. The responses to this question indicate that the participants had some difficulties understanding what information was expected to enter into the Wizard.

Participants provided 34 (M=3.4, SE=.6) and 42 (M=4.2, SE=.7) comments to their responses in the SUS and the FUS questionnaires, respectively. While overall usability scores are positive, the number of comments indicates that there may be some specific issues in the Wizard that should be further resolved. Finally, more comments in the FUS questionnaire indicate - as expected - that the FUS questionnaire was able to capture the issues that are specific to online forms.

UMUX results. The mean overall UMUX score is positive (M=72.5, *SD*=16.7, *SE*=4.3). The distribution of the overall scores also indicates positive evaluation with only one overall score being negative. The correlation of the UMUX scores with the SUS scores is significantly high (r(13) = .88, p < .001). The correlation of the UMUX scores with the FUS scores is significantly high as well (r(13) = .74, p = .002). The results indicate that the evaluations of all three usability components (effectiveness, satisfaction, efficiency) are (moderately) positive. The Wizard is fit for its purpose for managing metadata and artefacts. The participants reported positive efficiency indicating that asset and metadata management was fast and did not require substantial effort. Finally, the participants reported positive satisfaction towards using the Wizard.

Table 2. UMUX usability scores.

Usability component	Effectiveness	Satisfaction	Efficiency	Overall
Mean score **Standard error**	M=2.8 SE=.2	M=3.1 SE=.2	M=2.9 SE=.2	M=2.8 SE=.2

Questionnaire answers
Participants' comments collected from the open input fields in the questionnaires provided some more detailed qualitative feedback. Some of the issues that were identified:

- Data storage
  - Users may lose their data after going to a next step, as the data are not saved in each step.
  - Users will lose their data when mandatory data are not filled.
- Support
  - Some fields require more explanations and suggestions to become more understandable.
  - Some fields seem to overlap, or would need better explanation.
- Progress indicator
  - It is not clear why after completing an asset's data, the progress indicator shows a completion percentage by less than 100%.
  - If the progress score has become close to 100%, it is difficult to understand, which part still requires attention for completion.

Overall result

Given the usability scores on the diverse tests it can be concluded that the participants were moderately positive towards using the wizard. The wizard smoothly guides the user through the process of asset declaration and submission. Some improvements can be made by covering intermediate data storage, enhanced instructions and the significance of the progress indicator (cf. Annex 5).

Additional fine-tuning of the wizard and its instructions will be addressed in the next version of the wizard software.

Usability References

1. Brooke, J. (1996). SUS - A quick and dirty usability scale. Usability evaluation in industry, 189 (194), 4-7.
2. Aeberhard, A. (2011). FUS - Form Usability Scale. Development of a Usability Measuring Tool for Online Forms. Unpublished master's thesis. University of Basel, Switzerland.
3. Seckler, M., Heinz, S., Bargas-Avila, J. A., Opwis, K., & Tuch, A. N. (2014). Designing usable web forms: empirical evaluation of web form improvement guidelines. In Proceedings of the 32nd annual ACM conference on Human factors in computing systems (pp. 1275-1284). ACM.
4. Finstad, K. (2010). The usability metric for user experience. Interacting with Computers, 22 (5), 323-327.

5 ASSET METADATA VIEWER

When users access the metadata of software assets there are two situations – either the users are allowed to modify these data, or they are allowed to view them only. The RAGE Metadata Editor uses the same interface for both situations. The only difference is the lack of [Save] button for users without write permissions. The advantage of this approach is the reuse of the same tool for two different purposes – editing and viewing. The disadvantage is that users face the complexity of the metadata model even if they need just to view the metadata.

The purpose of the RAGE Wizard is to assist in preparing the metadata description of a software asset. It splits the efforts into several annotated steps. This style of presenting the metadata is not suitable for the case when users just want to view the asset. To access this problem we developed the RAGE Metadata Viewer. This tool extracts the metadata of a software asset and presents it in a structured page – see figure 4.

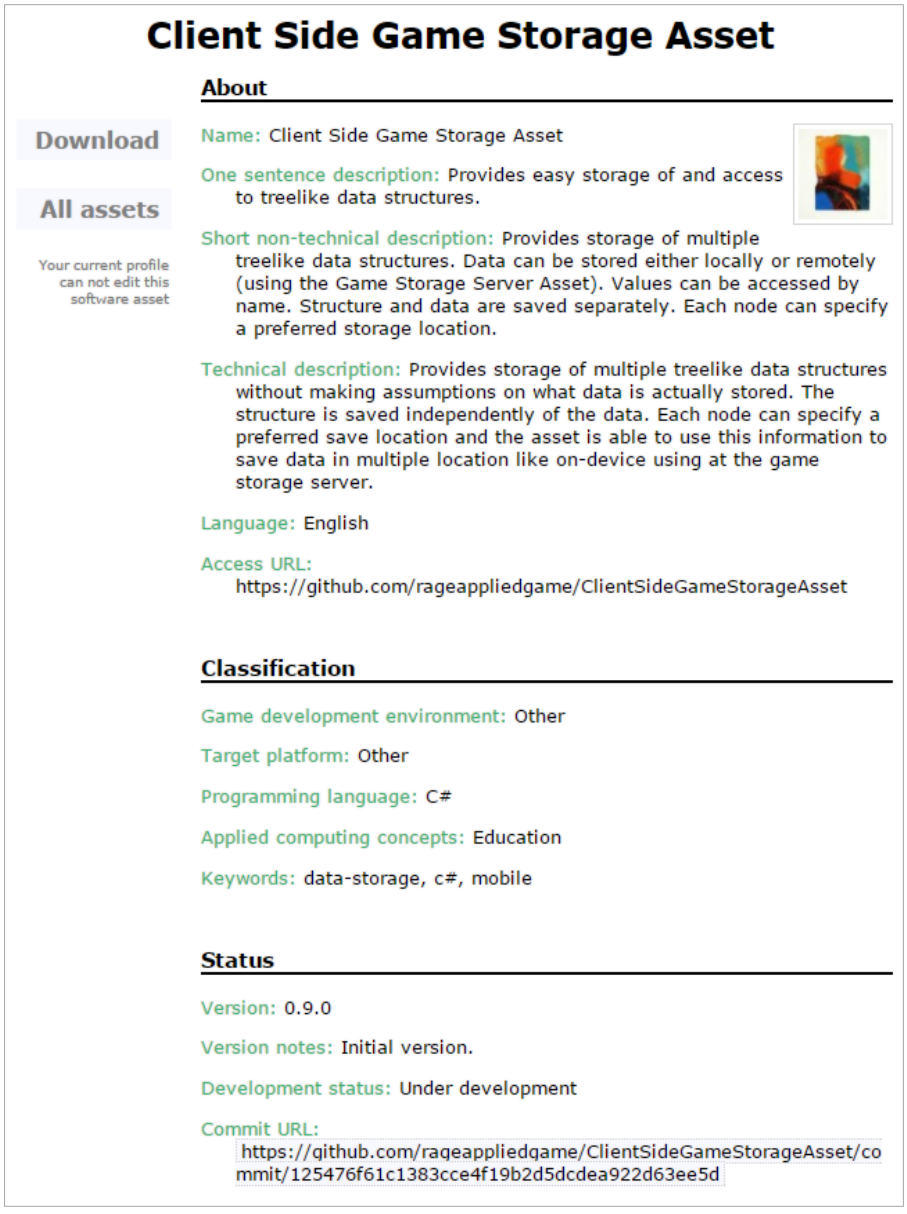


Figure 4. Screenshot example of the metadata viewer.

The main purpose of the viewer is to provide an easy-to-read description of a software asset with appropriately grouped metadata. Additionally, the viewer interface is printer-friendly – i.e. if the user prints the page, the viewer hides navigation and control buttons and prints only the metadata. This is suitable for creating off-repository records of assets.

The software asset viewer is the intermediate layer between the front-end asset manager and the asset wizard – Figure 5. Asset users browse and search all assets in the repository within the asset manager. When they click on a selected asset, it is opened in the asset viewer. Asset users can inspect the asset description and download it if they like to incorporate it in their game. However, if the users are the developers of this asset or if they have sufficient write permissions then can further open it in the wizard to edit it. When finished editing the asset, they automatically return to the viewer to review the changes.

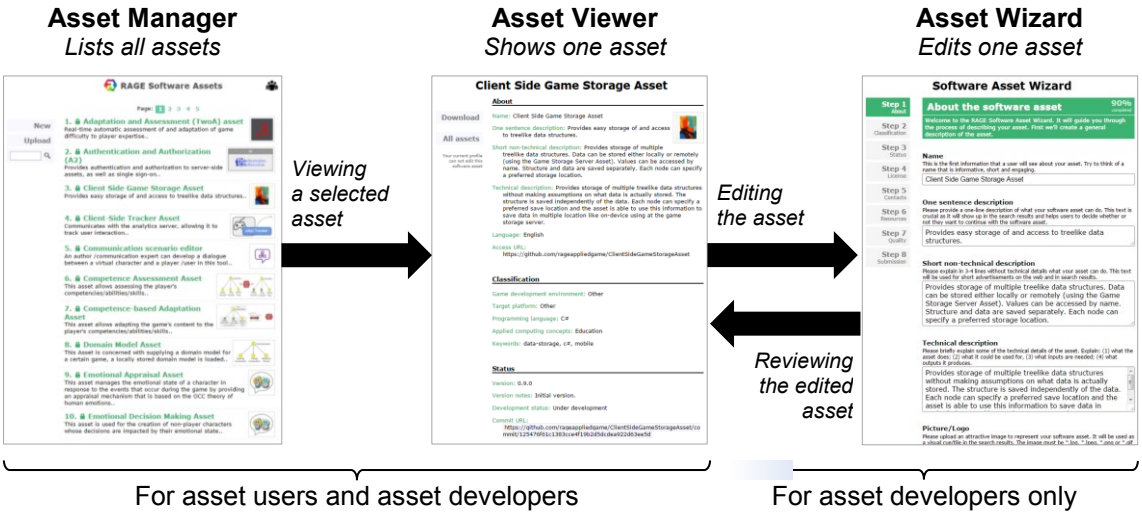


Figure 5. The asset viewer as intermediate layer between the asset manager and the asset asset creation wizard.

6 COORDINATION OF ASSET CREATION AND ASSET PUBLICATION (WP1/WP6)

6.1 Aligning WP1 and WP6 activities

This chapter reflects coordination of processes designed in WP1 (asset repository) and WP6 (ecosystem), respectively. It describes the principles and decisions needed for preserving correct inclusion of the asset repository in the ecosystem portal. The RAGE asset repository (WP1) is the asset creation environment that accommodates the declaration and submission of software assets and its constituents in accordance with RAGE's asset metadata model. It is one of the subsystems of the RAGE ecosystem portal (WP6), which is the asset publishing and community environment reaching out to end-users of the assets.
The description fits into the phased approach that was agreed upon with WP6, which entails initial focus on the metadata schema and asset repository (WP1) supporting the (mainly academic) asset creators. Now that the asset repository and its tools have become available the first level of asset consumers (software developers looking for published assets for re-use) and their interaction with the ecosystem portal comes into view an update will be needed of the ways these user groups interact with the different subsystems. A subsequent update will be required at a finale stage when different end-user groups will be considered.

Some key concepts are further explained below:

Asset (software asset)
In RAGE, the assets are composed of software component(s) and additional artefacts such as manuals, documentation, scientific evidence, examples of use, demos, content authoring tools and a wide range of additional resources, as well as the set of metadata to describe these (cf. RAGE glossary).

Asset repository
The asset repository (cf. WP1) is essentially a structured collection of assets, along with an authoring environment for defining and editing the assets. Asset creation entails the uploading of the asset's ingredients (artefacts) and specifying the associated metadata. End-users of the repository are (external) technology providers that want to expose their game software assets to the wider game industry and game research communities through the RAGE ecosystem portal.

Ecosystem portal
This system acts as a publishing and distribution environment that uses the assets (or their ingredients) to establish market offerings to target groups in the applied game industry and game research communities and accommodates a social space for these.

6.2 Starting points

- Exploitation of the RAGE system(s) should be lean and efficient, requiring a minimum of human interference and overhead costs.
- Asset submission to RAGE should be as easy as possible, given the general aversion of tech developers of specifying metadata, and the associated risk to loose contributors already during the submission process.
- RAGE will partly rely on well-established external services such as GitHub or Slideshare rather than trying to duplicate those services (which is out of scope).
- Accepting early software versions/prototypes for inclusion would be in agreement with common practices in IT communities, but should go with clear labelling with respect to software status.

6.3 Asset repository as a loosely coupled sub-system

The idea of defining the asset repository as a separate loosely coupled subsystem of the ecosystem portal rather than a single monolithic system is motivated by the extra flexibilities and

opportunities of content collection and content distribution it offers. First, by its independence the asset creation subsystem (the asset repository) would allow the exposition of assets to multiple third party publishing platforms. Second, the ecosystem portal allows for collecting additional content from external sources. Finally, asset creation is simply different from asset publishing, marketing and selling: the two functions require different contents, sometimes covered by different agents.

6.4 Asset creation

Primary users of the asset creation system are (external) tech providers, tech developers, or tech owners, all covered by a single user profile: asset creator. The final result of the asset creation process is the inclusion of the asset (viz. its artefacts and its metadata) in the RAGE repository. Submission to the repository does not imply the publication of the asset, as publication needs to be prepared and arranged in the RAGE ecosystem portal.
Upon accessing the repository, the asset creators can use an asset manager and asset viewer to inspect assets and an asset editor to create new assets or adjust existing ones. In order to not deter asset creators beforehand or during the asset creation process, we have radically simplified the process by including an asset creation wizard that guides the asset creator step by step toward asset submission. Also, we have defined a minimum set of mandatory metadata fields, which are clearly marked in the wizard. In the wizard the uploading of artefacts is directly collocated with specifying metadata. Artefacts can either be uploaded or referred to via an URI. Although the latter implies dependencies of external systems, e.g. CodePlex, BitBucket or GitHub for software commits, and thereby may lead to broken links when the external services are dismantled, the recreation of many services is beyond the scope of RAGE: e.g. RAGE cannot be expected to develop and replace a fully-flashed software versioning service as GitHub. If an external service would disappear, the asset creator has to recreate the asset by processing the changes (see also “synchronisation” below).

Quality assurance
So far, quality checks by RAGE can only be performed on the completeness of metadata. The asset wizard presents contextual instructions and explanations as to prevent mistakes. The quality of artefacts, e.g. the software, the documentation, the manuals, tutorials etc., are accepted as is. We might say that for the assets produced by RAGE manual inspection would be possible to make sure we start with a well-defined set, but during exploitation after the ending of the project labour overheads should be kept to a minimum. Therefore, quality assurance is based on self-declaration: upon submission, the asset creator is required to answer some questions, e.g. about software documentation, coding style, RAGE architecture compliancy, etc., and has to declare that all information provided is correct. This also touches on warranty and liability issues: the asset creator remains fully responsible. In addition, an important self-organised quality assurance instrument would be the community feedback that is collected in the ecosystem portal through quality ratings by end-users.

Allowing early version software
In accordance with the policies of software versioning systems (e.g. Github) early version software is accepted for inclusion without barriers as to allow for early feedback and involvement from community members. For indicating software maturity we will use the Semantic Versioning Specification 2.0.0 (SemVer, http://semver.org/’, which is explained in the asset wizard). Version zero software (0.y.z) is reserved for early development releases and patches: the public API should not be considered stable.

Asset packaging
Although the asset creation sub-system allows for packaging the asset for distribution, all asset’s ingredients (artefacts) are stored separately and are available for reuse in different constellations.

6.5 The ecosystem portal

The metadata of an asset and all of its ingredients (artefacts) become available in the ecosystem portal, which allows for preparing the enrichment, marketing and publication of

assets. Here it is determined how the asset shows up in the portal. Different user roles may be authorised for the publishing of assets. In many cases this will be the very same asset creator, but also more marketing and sales oriented users or even software publishers may be involved here, depending on permissions granted. The user role in charge for publishing is generically referred to as the "asset publisher".
The asset publishing process basically entails three processes: 1) the arrangement or rearrangement of the asset's ingredients/artefacts, 2) the enrichment of the assets with additional content, and 3) the act of making the asset available as a package in the portal.

1. (Re)Arrangement of the asset's artefacts
   The asset publisher will generally adopt the asset's core content as specified in the repository. But he/she may also use the wider pool of artefacts, e.g. arising from other assets posted under the same account, to adjust the composition of the asset, e.g. for tuning it to a specific target group.
2. Enrichment
   Three methods are available for enriching the asset with additional content:
   - Manual inclusion of additional content, e.g. harvested from external social media platforms (e.g. Mendeley, Slideshare).
   - Automated enrichment through a recommender service.
   - The inclusion of marketing, sales and pricing information.
3. Packaging and releasing
   This act entails pressing a button after a preview check.

Quality assurance
As is the case in the asset repository, quality assurance and liability issues are covered by a self-declaration approach, to be confirmed by the asset publisher upon publication. Community rating of assets in the portal are an important self-organised approach to monitor and improve asset quality.

Synchronisation of the asset creation and asset publishing
All data and artefacts entered in the asset repository will become available to the ecosystem portal through a one-off, unidirectional pipeline synchronisation. This means that the publishing service preserves the synchronised asset in its fixated composition, and remains ignorant of any posterior changes made in the repository. The main reason for this approach is to avoid potential software compatibility issues when a new software version would become available under the same asset ID. To deal with updates the original asset on the repository side should be cloned to another ID and adjusted; the same should be done in the ecosystem portal once the new data have become available. In the portal, different versions should be grouped together for their presentation.
This one-off synchronisation also radically reduces system overheads.

The single user scenario
A baseline use case would be a single user both creating and publishing the asset. The switch between the two subsystems should be seamless. This is enabled through a single-sign-on to preserve the same session, and a synchronised user interface that allows for a smooth, stealth transition.

6.6 Anticipated actions for follow-up

- Clarify the boundaries of the repository content
- Involve external tech providers for testing the asset creation and publication process
- Anticipate future changes of the metadata schema (based on usability test, and stakeholder consultations)
- Explore automated quality assurance services (e.g. checks and balances)
- Design a quality assurance procedure for processing community ratings
- Establish a seamless user interface shielding the transition between two subsystems
- Define terms and conditions; liability and warranty statements for assets and for the portal

- Updating of the ways different user groups (viz. game developers and wider user groups) interact with the different subsystems.

7 OUTLOOK

Asset developers have qualified the wizard as a usable and appropriate tool for annotating and uploading the assets. But this is not the end point of our activities. First of all, external technology providers may be less patient and dedicated to RAGE when entering their metadata. Therefore, in the period after completion of this deliverable, additional fine-tuning of the wizard and its instructions will be needed. For the internal launch of the RAGE ecosystem platform, which is foreseen in month 31-32, the wizard and metadata storage will be linked with the portal infrastructure. The quality of entered metadata and artefacts will be manually checked as to set high quality standards for the first batch of assets. Although the development of automated artefact quality checks, for example automated code checks, would be beyond the scope of the project, we think it worthwhile to explore some automation options that might help to enhance the quality of submissions. The style of the user interfaces will be aligned with the overall look and feel of the portal. Before the (soft) external launch of the ecosystem portal, which is scheduled in month 35-36, external parties will be involved to explore the system and make judgements about its usability. Also after the external launch user appreciations will be investigated. Third parties, e.g. IT research projects, will be actively engaged and invited to upload their technologies.

LIST OF ANNEXES

Annex 1: Gala Paper about the Rage repository infrastructure

Georgiev, A., Grigorov, A., Bontchev, B., Boytchev, P., Stefanov, K., Westera, W., Prada, E., Hollins, P. and Moreno Ger, P. (2016). The RAGE Advanced Game Technologies Repository for Supporting Applied Game Development. In R. Bottina, J. Jeuring and M. Veltkamp (Eds.), Proceedings of the 5th International Conference, GALA 2016, Utrecht, The Netherlands, December 5-7, 2016, (pp. 235-245). Cham: Springer International Publishing. doi=10.1007/978-3-319-50182-6_21.
Preprint available as open access at http://dspace.ou.nl/handle/1820/7329

An extended version of the manuscript will be published in the International Journal of Serious Games.

Annex 2: C# manual client-side RAGE assets

Annex 3: TypeScript manual client-side RAGE assets

Annex 4: Code review checks for client assets

Annex 5: Outcomes of the Asset Creation Wizard Usability Study

The RAGE Advanced Game Technologies Repository for Supporting Applied Game Development

A. Georgiev[1], A. Grigorov[1,6], B. Bontchev[1], P. Boytchev[1], K. Stefanov[1], W. Westera[2], R. Prada[3], Paul Hollins[4], Pablo Moreno[5]

[1] Sofia University "St. Kliment Ohridski", Faculty of Mathematics and Informatics, Bulgaria
{atanas,alexander.grigorov,bontchev,boytchev,stefanov}@fmi.uni-sofia.bg
[2] Open University of the Netherlands
Wim.Westera@ou.nl
[3] University of Lisbon, Portugal
rui.prada@tecnico.ulisboa.pt
[4] The University of Bolton, UK
pah1@bolton.ac.uk
[5] Universidad Complutense de Madrid, Spain
pablom@fdi.ucm.es
[6] Institute of Mathematics and Informatics, Bulgarian Academy of Sciences, Bulgaria
grigorov@math.bas.bg

Abstract. This paper describes the structural architecture of the RAGE repository, which is a unique and dedicated infrastructure that provides access to a wide variety of advanced technologies (RAGE software assets) for applied game development. These software assets are reusable across a wide diversity of game engines, game platforms and programming languages. The RAGE repository allows applied game developers and studios to search for software assets for inclusion in applied games. The repository is designed as an asset life-cycle management system for defining, publishing, updating, searching and packaging for distribution of these assets. The RAGE repository provides storage space for assets and their artefacts. It will be embedded in a social platform for networking among asset developers and other users. A dedicated Asset Repository Manager provides the main functionality of the repository and its integration with other systems. Tools supporting the Asset Manager are presented and discussed. When the RAGE repository is in full operation, applied game developers will be able to easily enhance the quality of their games by including advanced game technology assets.

Keywords: software assets, serious games, asset repository, asset development, taxonomy tools, metadata editor, applied games, reuse.

1 Introduction

Applied gaming is highlighted as one of the main priorities in Horizon2020, the Research and Innovation Programme of the European Commission. Policy makers of the

European Commission envision a flourishing applied games industry that helps to address a variety of societal challenges in education, health, social cohesion and citizenship, and equally one that stimulates the creation of jobs in the creative industry sector.

Although applied or serious games have been successfully employed in education and training settings across a wide and varied range of application domains, seizing the full potential of applied games has been challenging. In contrast, the leisure games industry is an established industry dominated by large international hardware vendors (e.g. Sony, Microsoft and Nintendo) and large publishers and retailers. Conversely, the applied game industry is fragmented across a large number of small independent businesses with limited interconnectedness and knowledge exchange [1, 2].

The RAGE project [3] aims to stimulate the applied game industry by making available a set of advanced reusable game technology components (software assets) that game studios can easily integrate in their game development projects. Applied game studios would benefit from using state-of-the-art technologies, while incorporating complex pedagogic technical functionality would become easier and quicker, and the cost of development would be reduced. The software assets cover a variety of functionalities including game analytics, emotion recognition, assessment, personalised learning, game balancing and player-centric adaptation, procedural animation, language technologies, interactive storytelling, and social gamification.

While the main research goal of the RAGE project is to support the applied game industry by making available a large set of reusable, advanced software components (applied gaming assets), this paper focuses on the design of the repository infrastructure that supports the processes of development, reuse and sharing of applied gaming assets. This paper presents the asset repository architecture and the associated asset development methodology. We first present the related work efforts, then discuss our approach (research method), describe the software asset concept, provide details of the design and implementation of the back-end repository system architecture and corresponding front-end tools, and we conclude with a brief description of first experiments with the infrastructure, analysis and identification of further development and research efforts.

2 Related work

Asset-based software development relies on reusing well documented and cohesive software artefacts and, therefore, it is inconceivable without a platform for storing and accessing assets. An asset repository as a software tool is defined by Ackerman and colleagues [4] for storing and retrieving reusable assets and managing asset access control for asset producers and consumers, according to the phases of the asset life cycle. They introduce the IBM Rational Asset Manager (RAS) repository, which handles tasks and activities of software asset producer, consumer and subscriber roles, while offering reduced production costs and improved software quality. In order to facilitate cross-project reuse of assets, the Rational Asset Manager model provides monitoring of asset categorization and usage together with multi-platform compliance management.

Another example for a RAS-based asset repository is the Atego Asset Library [5], which is a scalable Web-based repository for reusable software engineering artefacts. It is based on OMG RAS and integrates Unified Modelling Language (UML) and Systems Modelling Language (SysML) in order to facilitate asset reuse at design time.

Currently, the tool is supported as PTC Integrity Asset Library[1] and, besides the publishing, finding and reuse of assets, provides services as interest registry and notification, automatic file interrogation, traceable links and reuse metric dashboard.

Extensions of the OMG RAS have been proposed for designing open source Web-based asset repositories providing advanced classification, search and utilization of reusable software assets of various types. The OpenCom asset repository was created as a supporting tool of Shanghai Component Library [6] based on an extension of OMG RAS profile aiming at collaborative creation of knowledge by web users. The Lavoi free source asset repository [7] was developed based on an extension of the component profile of OMG RAS broadening the categories about classification, solution, usage and related assets.

Within the computer games domain, the *asset* concept is often reserved for media files to be incorporated in a game. For example, the Intel® XDK HTML5 Cross-platform Development Tool [8] offers an asset manager for game development in conjunction with several game platforms. Here assets are often considered audio-visual game objects to be included in a project. In RAGE the focus is on software assets, reusable components adding specific (pedagogic) functionality for applied game development.

A similar attempt related to using a digital repository of metadata resources for education, combined with a portal for the respective community of practices build around the repository, is described in [9]. Other approaches to endowing digital libraries with adaptability capabilities in order to scaffold and enhance end user experience are presented in [10]. Similar attempts inside GALA Network of Excellence are the SoA framework for SGs [25] and the repository for exchange of game resources [26].

3 RAGE Software Assets

A RAGE asset as a self-contained software component related to computer games, intended to be reused and or repurposed across different game platforms. Its formal definition is compliant with the asset definition of the W3C ADMS Working Group [11], which refers to abstract entities that reflect some “intellectual content independent of their physical embodiments”. In principle, not all assets are required to include software, however this paper focusses on software assets.

The RAGE asset is designed to contain advanced game technology (software), as well as value-adding services and attributes that facilitate their use, e.g. instructions, tutorials, examples and best practices, instructional design guidelines, connectors to major game development platforms, test plans, test scripts, design documents, data capacity, and content authoring tools/widgets for game content creation.

[1] http://www.ptc.com/model-based-systems-engineering/integrity-modeler/asset-library

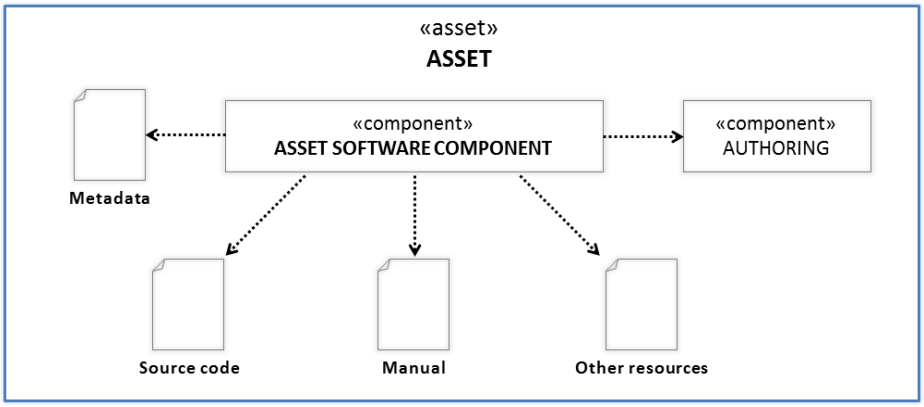


Figure 1. Conceptual layout of a RAGE Asset

Figure 1 presents the general layout of a RAGE asset. Its software architecture is component-based and has been described and validated in [12]. It addresses both the internal workings of an asset and the level of interaction of assets with the outside world, including the mutual communications between assets. The RAGE architecture avoids dependencies on external software frameworks and minimises code that may hinder integration with game engines. It relies on a limited set of standard software patterns and well-established coding practices. Each RAGE asset contains metadata, which describe its content and functionality. RAGE metadata model in the domain of applied gaming was designed for defining the asset's metadata and for enabling the proper implementation of the RAGE Asset repository system architecture [13].

4 Our approach

The research methodology for this study is based on the Rapid Application Development model [14]. We performed an extensive needs assessment study [15], including asset developers, educators and game producers. We have identified the services to be supported through the repository and other related tools and, in parallel, designed the RAGE metadata model to fit the specified domain of reusable gaming components (RAGE software assets). It was clear that we could not reuse any existing solution, but needed to design and implement our own software repository, targeting the identified needs and characteristics of the applied game domain.

In the next stage we provided the initial design of the RAGE asset as a software component, and the architecture of the RAGE software repository, aimed at supporting the development, storage, sharing and reuse of assets. In the next stage we provided details on the technical implementation of the software repository. We performed several interactions between these two stages until we reached a stable and more or less complete solution. In the last stage we analysed the first use case scenarios of the repository through several client tools, arranged first evaluations of the repository, and collected ideas for its improvement in the next cycle.

We will present the results of each stage in the next sections.

5 The Asset repository system architecture

Metadata is a key part of the information infrastructure necessary to help create order and provide a solid foundation for providing various information services such as descriptions, classifications, organizations, store, search, creation, modification and aggregation of information [16]. Rather than merely a software archive, the asset repository is viewed as a system for managing the lifecycle of an asset. In the repository the asset's artefacts are collected and conceptually tied together by defining the metadata. In addition, the repository allows for publication, updating, packaging for distribution and quality assurance, while accommodating different end-user tools.

The RAGE asset software repository is at the core of the asset development infrastructure. It is used to store and manage access to: (1) reusable game assets, (2) artefacts (resources within game assets), (3) metadata for game assets and artefacts, and (4) relationships between assets – dependencies, related assets, etc.

The Asset software repository leverages the discovery, development reuse and repurpose of game assets and artefacts. It will help both game asset developers and consumers in all the activities relating to the game asset lifecycle.

The main functions of the RAGE Asset software repository are as follows:

- Searching, finding and browsing assets/artefacts
- Creating, updating, publishing, deleting and downloading assets/artefacts
- Versioning support, source code import from GitHub and integration with IDEs
- Harvesting of external repositories for game assets and metadata using the Open Archives Initiative - Protocol for Metadata Harvesting (OAI-PMH)
- Reviewing and rating assets/artefacts

In order to implement these functions, we designed the asset repository infrastructure in three tiers (Figure 2): client, service and data store tiers.

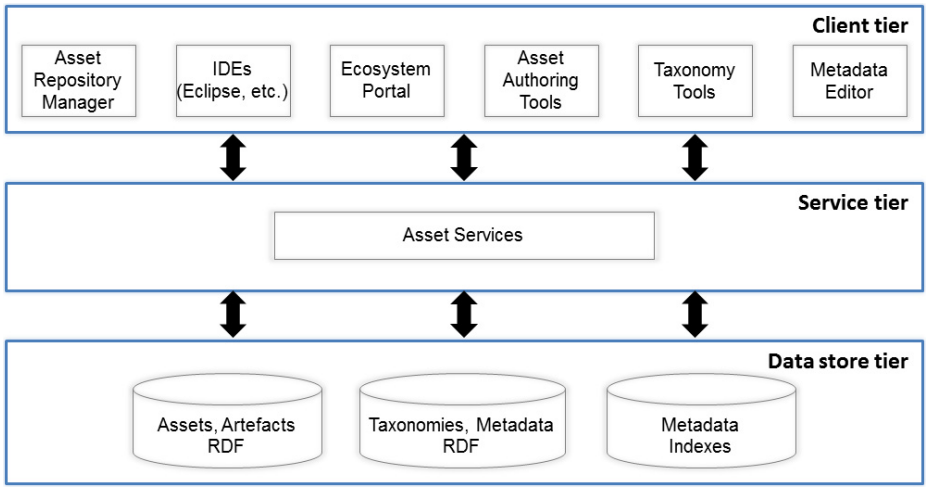


Figure 2. Asset Repository Architecture

6 Implementation of the asset repository system architecture

The main result from the second stage – Acting, is the implementation of the Asset repository. Fedora [17] is used for storing assets, metadata and artefacts; Sesame [18] for managing RDF data and supporting classification and entities; and Solr [19] for indexing and searching the repository. The data store tier consists of these three components and is used to store game assets, artefacts, metadata, taxonomies and indexes:

- **Fedora** stores the game assets, artefacts and metadata using RDF as primary data format. When the repository is updated by creating, modifying or deleting resources, it generates specific events so that the Fedora indexer copies RDF from the repository to an external triple store to keep it synchronized with the repository. Fedora is flexible, well established and it ensures scalability and durability (the complete repository can be rebuilt at any time).
- **Sesame** is an architecture for the efficient storage and expressive querying of large quantities of metadata in RDF and RDF Schema. This includes creating, parsing, storing, inferencing and querying over such data. Sesame RDF triple store contains metadata from Fedora and classification taxonomies/vocabularies.
- **Solr** is an open source platform optimized for searching. Its major features are full-text search, sophisticated faceted search, almost real-time indexing, dynamic clustering of data, etc. It is used for creating full text indexes on the RAGE metadata fields, as well as for realizing full text search and faceted search.

The service tier is used for access and preservation of the assets and artefacts. For the implementation of this tier, we developed the following services that provide access to the underlying data store tier:

- **Fedora Services**. Fedora provides a general RESTful HTTP API for accessing repository resources through HTTP methods. It supports OAI-PMH [20] requests on content and metadata in the repository.
- **Sesame Services**. Sesame offers a RESTful HTTP interface supporting the SPARQL Protocol for RDF. It is a superset of the SPARQL and supports communication for Update operations and the Graph Store HTTP Protocol [21].
- **Solr Services**. Apache Solr exposes Lucene's Java API as REST-like API's which can be called over HTTP. The RESTful endpoints allow CRUD style operations to be performed on the repository resources.

In addition, for the service tier to provide access to the client tier, we developed **Asset Services** for composition and execution of workflows over RAGE Game Assets.

The client tier includes web-based applications, plug-ins for integrated development environments (IDEs), and software components from the RAGE ecosystem that uses the services supported by Asset Repository Infrastructure. It includes:

- **The Asset Repository Manager** – we developed a web-based application embodying main functionalities for lifecycle management of assets and artefacts.
- **IDE plug-ins** – we developed rich clients consuming services from the Asset Repository service tier, which thus allows developers to manage assets from within their integrated development environment (IDE).

- **Other software components from the RAGE ecosystem**, such as the Ecosystem Portal (EP), which harvests assets and metadata through an OAI-PMH service provider from Asset Repository Service tier.

The Asset Repository services constitute an open interface for creating, modifying, deleting, and searching RAGE assets. They are realised on top of REST APIs, JSON, JSON-LD [22] and RDF, using Software as a Service (SaaS) model in the cloud. Based on the functionality exposed by these services, they can be grouped as:

- **Asset Access Services** defining an open interface for accessing assets within the RAGE Asset Repository allow for retrieving asset packages and metadata, and to search and browse for assets using keywords and metadata fields. The search interface provides both full-text search and semantic search. Full-text search enables performing of natural language queries using keywords and phrases occurring in any of indexed asset's metadata elements. The semantic search is using SPARQL for querying on asset metadata and SKOS taxonomies data represented as RDF triples.
- **Asset Management Services** defining an open interface for administering assets, including creating, modifying, and deleting, provide an abstract level of the operations, thus hiding the complexities of the internal formats, protocols and procedures for storing an asset in the Asset Repository.
- **Taxonomy Services** defining an open interface for managing classification taxonomies and controlled vocabularies used in RAGE Asset Metadata Model [13] to classify and describe an asset in educational and gaming contexts. For representation and storing Asset Repository adopts SKOS standard [23].
- **Authentication and Authorization Services** provide access for organisational needs. These services are implemented on top of Fedora Authentication and Authorization framework [17].

7 Usage scenarios

In order to observe how the asset repository together with related client tools can support the asset developers and other users, and how effective and useful the services are, which it is offering, we have designed various usage scenarios. Also, asset developers and game developers have been involved for evaluating the functioning and usability of the repository. In this section we will present the scenarios, and in the next section will present the main conclusions based on the observations of real users.

To populate the repository with metadata we used four usage scenarios. The first scenario is publishing/updating a game asset through the web-based interface offered from the Asset Manager. The asset developer signs in, creates/selects an asset, enters/updates metadata and uploads artefacts or a packaged asset (see Figure 3).

The second scenario is publishing/updating a game asset from GitHub. The asset developer again should sign in the Asset Manager, creates/selects an asset, provides the GitHub repository identifier and credentials (if required). The files (artefacts) and metadata from GitHub are automatically harvested and published in the RAGE Asset

Repository (using the GitHub API [24]). The user should also supply the rest of the required metadata.

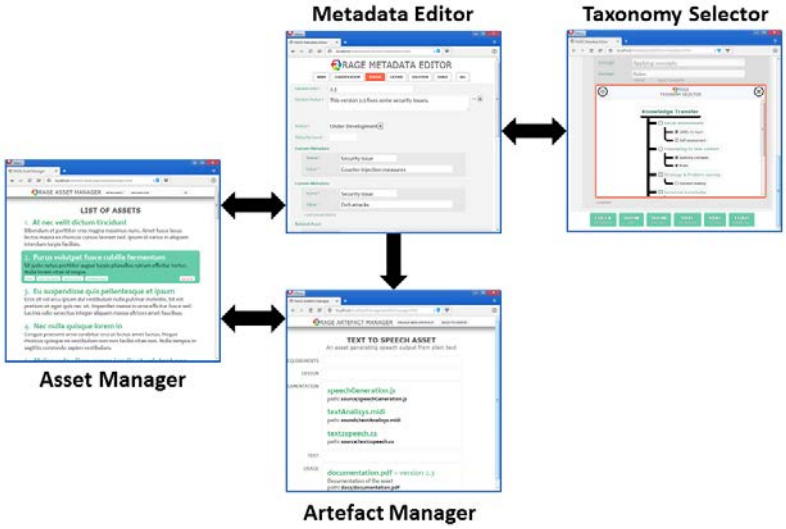


Figure 3. Using the RAGE Asset and Artefact managers, the RAGE Metadata editor and the RAGE Taxonomy selector to populate the repository

In the third scenario, we tested publishing/updating a game asset from an IDE. For this scenario we developed an Eclipse IDE plugin. The asset developer opens the asset project in the Eclipse IDE; using the plugin the developer creates/updates the asset in RAGE Asset Repository within the IDE, providing credentials and needed metadata.

The fourth scenario: Asset consumers can search for a game asset using full text or advanced search, browse the repository, view assets metadata and download assets or artefacts for reuse.

At the moment, the repository is populated with the metadata of 12 currently developed Assets in RAGE project.

8 Scenario evaluation

An evaluation of the usage scenarios was carried out by involving a group of 9 end users, viz. asset developers from the RAGE project. Preliminary findings of this user panel support the relevance of the repository system. Comments about the first version of the repository and related client tools can be summarized as follows:

- Users can easily work with basic services such as searching, downloading or uploading assets to the repository.
- Users need more specific instructions how to populate the repository with metadata.
- The metadata editor improved the process of populating the repository for users.

- Users encounter problems to identify the source of the information related to some of the metadata fields, like keywords and others.
- There is a need to automate further the definition of metadata fields.

While the evaluation is preliminary and relatively informal, the initial acceptance is positive, and confirms the viability of this first step within the RAGE Project.

9 Conclusions and future work

In this paper, we presented a unique software architecture supporting the lifecycle of reusable software components for applied gaming. The main innovation is related to the combination of RAGE Asset Model and RAGE Asset Metadata Model, backed up with server-side infrastructure (repository and services) and many end user tools. The software architecture plays a pivotal role within the RAGE Ecosystem, developed for the RAGE project and is considered of strategic importance for the domain of applied gaming.

The repository as the content core system of the RAGE Ecosystem allows for flexible design and development of RAGE game assets and future search, packaging and exchange. The current architecture guarantees both scalability and durability and the approach. It also provides a high level of flexibility across different taxonomies and standards.

Future work is planned on improving the architecture by providing support for Quality Assurance, asset development workflows, harvesting of assets from external systems and stores, social functions and for specific targeted support for the gaming community. A first provisional launch of the repository integrated in the RAGE social platform is expected in 2017.

Acknowledgements. This work has been partially funded by the EC H2020 project RAGE (Realising an Applied Gaming Eco-System); http://www.rageproject.eu/; Grant agreement No 644187.

References

1. García Sánchez, R., Baalsrud Hauge, J., Fiucci, G., Rudnianski, M., Oliveira, M., Kyvsgaard Hansen, P., Riedel, J., Brown, D., Padrón-Nápoles, C.L., Arambarri Basanez, J.: Business Modelling and Implementation Report 2, GALA Network of Excellence, www.galanoe.eu.
2. Stewart, J., Bleumers, L., Van Looy, J., Mariën, I., All, A., Schurmans, D., Willaert, K., De Grove, F., Jacobs, A., Misuraca, G.: The Potential of Digital Games for Empowerment and Social Inclusion of Groups at Risk of Social and Economic Exclusion. Joint Research Centre, European Commission, Brussels. http://ftp.jrc.es/EURdoc/JRC78777.pdf (2013)
3. RAGE: Project Web site (2015) http://www.rageproject.eu .
4. Ackerman, L., Elder, P., Busch, C.V., Lopez-Mancisidor, A., Kimura, J., Balaji, N.A.: Strategic reuse with asset-based development, IBM RedBooks (2008) http://www.redbooks.ibm.com/redbooks/pdfs/sg247529.pdf
5. Kattau, S.: Atego launches RAS-based asset repository, SD Times Magazine, February 13, 2013, http://sdtimes.com/atego-launches-ras-based-asset-repository/#ixzz3wwMlvLJ8

6. Hong-min, R., Zhi-ying, Y., Jing-zhou, Z.: Design and Implementation of RAS-Based Open Source Software Repository, Proc. of te Sixth Int. Conf. on Fuzzy Systems and Knowledge Discovery, Vol.2, pp.219-223 (2009).
7. Moura, D. S.: Software Profile RAS: estendendo a padronização do Reusable Asset Specification e construindo um repositório de ativos, Master's thesis, Univ. Federal do Rio Grande do Sul, Brasil (2013) http://www.lume.ufrgs.br/handle/10183/87582
8. Hilliar, G.: Developing Cross-Platform Mobile Apps with HTML5 and Intel XDK, in Dr. Dobb's Journal, UBM plc. (2014)
9. Böhm, T., Klas, C.-P., Hemmje, M.: Supporting Collaborative Information Seeking and Searching in Distributed Environments. In Proc. Of the LWA 2013 Conference, Bamberg, Germany, pp 16-20 (2013).
10. Stefanov, K., Nikolov, R., Boytchev, P., Stefanova, E., Georgiev, A., Koychev, I., Nikolova, N., Grigorov, A.: Emerging Models and e-Infrastructures for Teacher Education, 2011 International Conference on Information Technology Based Higher Education and Training ITHET 2011, IEEE Catalog Number: CFP11578-CDR, ISBN: 978-1-4577-1671-3.
11. Dekkers, M.: Asset Description Metadata Schema (ADMS). W3C Working Group (2013)
12. Van der Vegt, G.W., Westera, W., Nyamsuren, N., Georgiev, A., Martinez Ortiz, I.: RAGE architecture for reusable serious gaming technology components, International Journal of Computer Games Technology, Vol 2016 (2016), http://dx.doi.org/10.1155/2016/5680526 .
13. A. Georgiev, A. Grigorov, B. Bontchev, P. Boytchev, K. Stefanov, K. Bahreini, E. Nyamsuren, W. van der Vegt, W. Westera, R. Prada, P. Hollins, P. Moreno. The RAGE Software Asset Model and Metadata Model, Serious Games, 2nd Joint Int. Conference, JCSG 2016, Springer, V. 9894 Lecture Notes in Computer Science, pp. 191-203, 2016.
14. Martin, James: Rapid Application Development, Macmillan, 1991.
15. Hollins, P. Westera,W. Manero Iglesias, B.: Amplifying applied game development and uptake, In Proceedings of 9th European Conference on Game-Based Learning ECGBL 2015, pp. 234-241, Steinkjer, Norway (2015)
16. Duval, E., Hodgins, W., Sutton, S., Weibel, S. L.: Metadata principles and practicalities. D-lib Magazine, 8(4), DOI: 10.1045/april2002-weibel (2002).
17. Woods, A.: Fedora 4.3 Documentation https://wiki.duraspace.org/display/FEDORA43/
18. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. First International Semantic Web Conference, Lecture Notes in Computer Science, pp 54--68, Springer Verlag (2002).
19. Smiley, D., Pugh, E., Parisa, K., Mitchell, M.: Apache Solr 4 Enterprise Search Server, Packt Publishing, ISBN: 9781782161363 (2014).
20. Lagoze, C., Van de Sompel, H.: The Open Archives Initiative Protocol for Metadata Harvesting (2015) https://www.openarchives.org/OAI/openarchivesprotocol.html
21. SPARQL 1.1: SPARQL 1.1 Overview, W3C Recommendation (2013)
22. JSON-LD 1.0: A JSON-based Serialization for Linked Data, W3C Recommendation (2014)
23. SKOS: Simple Knowledge Organization System Reference, W3C Recommendation (2009)
24. GitHub API: GitHub Developer Guide (2016) https://developer.github.com/v3/\
25. M. B. Carvalho, F. Bellotti, R. Berta, A. De Gloria, G. Gazzarata, J. Hu, M. Kickmeier-Rust: A case study on Service-Oriented Architecture for Serious Games, Entertainment Computing 6(2015), pp. 1-10, DOI:10.1016/j.entcom.2014.11.001
26. A. Gloria, F. Bellotti, R. Berta, and E. Lavagnino, "Serious Games for Education and Training," International Journal of Serious Games, Vol. 1, No. 1, 2014, pp. 100-105, ISSN: 2384-8766

APPENDIX 2

RAGE

Realising and Applied Gaming Ecosystem

Research and Innovation Action

Grant agreement no.: 644187

WP1 – Creating RAGE client-side assets in C#

RAGE – WP1

FINAL

Project Number	H2020-ICT-2014-1
Due Date	
Actual Date	26-June-2017
Document Author/s	Wim van der Vegt, Wim Westera
Version	1.1
Dissemination level	
Status	Final
Document approved by	

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644187

Document Version Control			
Version	**Date**	**Change Made (and if appropriate reason for change)**	**Initials of Commentator(s) or Author(s)**
0.1	September 2015	Initial version	WimV
0.2	October 2015	Revision after input OUNL	WinV, WimW
0.3	January 2016	Added C# Project Templates	WimV
1.0	08-May-2017	Updated Document	WimV
1.1	26-June-2017	Updated Document	WimV

Document Change Commentator or Author		
Author Initials	**Name of Author**	**Institution**
WimV	Wim van der Vegt	1.OUNL
WimW	Wim Westera	1.OUNL

Document Quality Control			
Version QA	**Date**	**Comments (and if appropriate reason for change)**	**Initials of QA Person**

Table of Contents

Creating C# Assets

For testing the practicability of the RAGE architecture a Proof-of-Concept asset was developed. The code of this Proof-of-Concept asset is used as a starting point for creating new assets. It references the RageAssetManager. The source code can be found in the GitHub repositories '**asset-proof-of-concept-demo-CSharp**' and '**AssetManager**' at https://github.com/rageappliedgame.

The code in the GitHub AssetManager repository consists of two Visual Studio projects.

1. The RageAssetManager project contains the actual code and targets the .Net 3.5 framework and can be used in for example Unity3D.
2. The RageAssetManager_Portable contains no sources of its own as it uses sources linked to the RageAssetManager project and targets a different .Net framework and generates a portable assembly usable in e.g. Xamarin.

Likewise the asset-proof-of-concept-demo-CSharp repository also contains two projects (a .Net 3.5 and a portable project) with some simple assets that were used for testing. The portable project again links to the sources of the .Net 3.5 project. Each of the projects links to the corresponding RageAssetManager project.

Creating an asset can be done in two ways. The more automated method is to use Visual Studio project templates, but it is of course also possible to create assets from scratch using the code of the asset-proof-of-concept-demo-CSharp repository at GitHub as starting point.

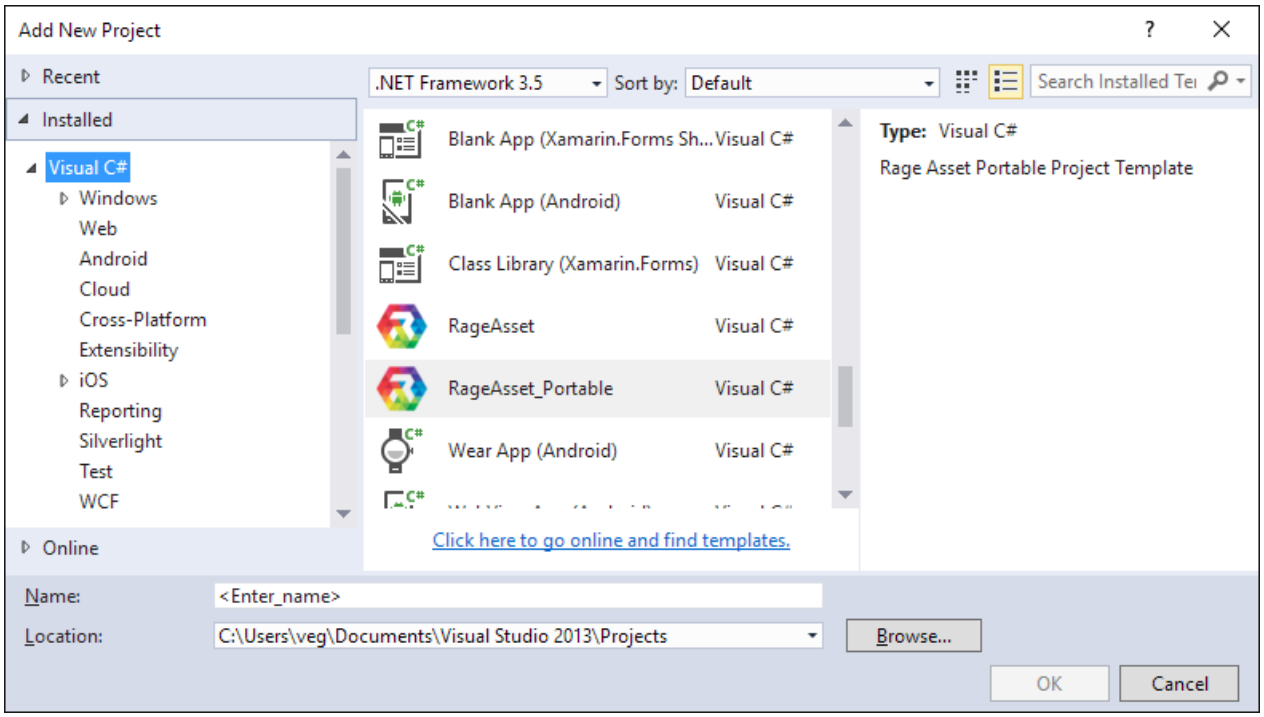
Asset Creation using templates.

For this method, two zipped templates have to be downloaded from the GitHub 'AssetManager-Project-Templates' repository at https://github.com/rageappliedgame. These zip files must be placed in the correct Visual Studio template folder.

In a default Visual Studio 2015 installation, the template base folder is a sibling of the project folder (where code is stored). By default, for Visual Studio 2015, the folder for placing the template files is '**Visual Studio 2015\Templates\ProjectTemplates\Visual C#**' under **My Documents**.

Remarks:
- The zipped templates should not be unzipped.

With the templates at the correct location, **File|New Project** will show two new C# project types: **RageAsset** and **RageAsset_Portable**.



First create a RageAsset project (the project name will be the name of the asset as well).

In this project correct the reference to the RageAssetManager assembly and the project can be compiled.

After that it is a good practice to create a RageAsset_Portable project as well inside the same solution (name it as the previous project with a '_Portable' suffix).

In this project one has to correct the reference to the RageAssetManager project as well. This portable project is setup with links to the sources in the RageAsset project (so when coding and subsequently compiling both projects, only a single set of sources is used).

Due to a 'bug' in Visual Studio the locations of the linked files have to be adjusted (they are relative to the temporary directory where the template is unzipped during processing). Basically open the portable project's csproj file with a text editor and correct the **Compile** and **Content** tags.

Tags like:

```
    <Compile
Include="..\..\..\..\..\AppData\Local\Temp\kwnaqsvc.4kc\MyFirstAsset\b.cs">
      <Link>MyFirstAsset.cs</Link>
    </Compile>
```

should be changed into:

```
    <Compile Include="..\TrackerAsset\TrackerAsset.cs">
      <Link>TrackerAsset.cs</Link>
    </Compile>
```

Besides correcting the include path, occurrences of **MyFirstAsset** in the example above have been replaced with the name of the previously created non-portable project (`TrackerAsset` in this example).

-Please note: Editing the project file in Visual Studio is also possible if you unload the project first, edit it and reload it.

-Please note: When adding new files to the non-portable project they have to be added as linked sources to the portable project as well. The context menu **Add|Existing Item** has a dropdown button that has an option to select **Add as Link**.

If the projects are setup as described, compiling them will immediately show.NET 3.5 vs Portable coding issues. If the projects compile correctly, two separate assemblies will be available for Unity3D and other platforms, e.g. Xamarin, respectively.

-Please note: The portable project needs a **PORTABLE** symbol defined in its Build tab.

Instead of referencing the RageAssetManager assembly, it is also possible to add the two RageAssetManager projects to the solution and make a reference to these projects.

-Please note: when using pdb2mdb utility, keep in mind that it requires the compiled assembly (the dll) as a parameter and not the pdb symbol file. The pdb2mdb utility converts .Net symbol files (pdb) into mono symbol files (mdb) that can be used with Unity3D to enhance debugging.

Manual Asset Creation

This method creates a non-portable asset project from scratch.

- For creating a new asset in Visual Studio, first create a blank solution and add the RageAssetManager project as an existing project.
- Add a new Class Library project that will contain your asset to be developed (from now on this will be referred in this manual as the MyAsset project).
- In the MyAsset project make a reference to the RageAssetManager project.
- Change the main class of the MyAsset project to:

```
namespace MyNameSpace
{
   using System;
   using System.Collections.Generic;
   using System.Linq;
   using AssetManagerPackage;
   using AssetPackage;

   /// <summary>
   /// An asset.
   /// </summary>
   public class MyAsset : BaseAsset
```

```
  {
    #region Fields

    /// <summary>
    /// Options for controlling the operation.
    /// </summary>
    private MyAssetSettings settings = null;

    #endregion Fields

    #region Constructors

    /// <summary>
    /// Initializes a new instance of the MyAsset class.
    /// </summary>
    public MyAsset()
      : base()
    {
      //! Create Settings and let its BaseSettings class assign Defaultvalues where it can.
      //
      settings = new MyAssetSettings();
    }

    #endregion Constructors

    #region Properties

    /// <summary>
    /// Gets or sets options for controlling the operation.
    /// </summary>
    ///
    /// <remarks>   Besides the toXml() and fromXml() methods, we never use this property but use
    ///             it's correctly typed backing field 'settings' instead. </remarks>
    /// <remarks> This property should go into each asset having Settings of its own. </remarks>
    /// <remarks>   The actual class used should be derived from BaseSettings (and not directly from
    ///           ISetting). </remarks>
    ///
    /// <value>
    /// The settings.
    /// </value>
    public override ISettings Settings
    {
      get
      {
        return settings;
      }
```

```
        Set
        {
            settings = (value as MyAssetSettings);
        }
    }

    #endregion Properties

    #region Methods

    // Your code goes here.
    // Try to keep only API code to be used by the Game-Engine here
    // and put all other code in separate classes.

    #endregion Methods
  }
}
```

Remark:

-When using the Settings property in the Game Engine code you will have to typecast it correctly to MyAssetSettings.

- Next, add a second class called MyAssetSettings to the MyAsset project that will contain the assets settings:

```
namespace MyNameSpace
{
  using System;
  using System.ComponentModel;
  using System.Xml.Serialization;
  using AssetPackage;
  /// <summary>
  /// An asset settings.
  ///
  /// BaseSettings contains the (de-)serialization methods.
  /// </summary>
  public class MyAssetSettings : BaseSettings
  {
    /// <summary>
    /// Initializes a new instance of the MyAssetSettings class.
    /// </summary>
    public MyAssetSettings()
```

```
    : base()
{
    //
}
/// <summary>
/// Gets or sets the test property.
/// </summary>
///
/// <value>
/// The test property.
/// </value>
[DefaultValue("Hello Default World")]
[XmlElement()]
public String TestProperty
{
    get;
    set;
}

/// <summary>
/// Gets the string[].
/// </summary>
///
/// <value>
/// .
/// </value>
[XmlArray()]
[XmlArrayItem("ListItem")]
[DefaultValue(new String[] { "Hello", "List", "World" })]
public String[] TestList
{
    get;
    set;
}
/// <summary>
/// Gets a value indicating whether the test read only.
/// </summary>
///
/// <value>
/// true if test read only, false if not.
/// </value>
public Boolean TestReadOnly
{
    get
    {
```

```
            return true;
        }
    }
  }
}
```

Remarks:

-The properties TestProperty, TestList and TestReadOnly are just example code and should be removed in the final code.

-Using statements and referenced assemblies should be kept minimal to avoid unnecessary dependencies.

- Create a Resources folder in the MyAsset project and include the following file called MyAsset.VersionAndDependencies.xml:

```
<?xml version="1.0" encoding="utf-8" ?>
<version>
 <id>myasset</id>
 <major>1</major>
 <minor>2</minor>
 <build>3</build>
 <revision></revision>
 <maturity>alpha</maturity>
 <dependencies />
</version>
```

  This file can later be edited to add dependencies to other assets and correct the version number and maturity (see the proof of concept code for how to specify dependencies).

- Now the asset can be compiled and can be used.
- For creating an instance of the asset (and have it create the RageAssetManager and register itself) add the following line to your game-engine code:

```
MyAsset asset = new MyAsset()
```

- The code below will return a textual report of the assets registered, their versions and (un)solved dependencies:

```
AssetManager.Instance. VersionAndDependenciesReport
```

Remarks:

-The settings base class will (if possible and specified) apply DefaultValue attributes to the properties so that the values are actually set (usually this attribute is only used by the .NET PropertyGrid to show when values are default).

-The asset base class will use the build-in XmlSerializer class to serialize and deserialize the settings to and from XML. It is therefore important that the MySettings class can be serialized. To influence the serialization various attributes can be added.

Creating bridges

A bridge can be added to both the MyAsset and the RageAssetManager.

- To persist (save/load) the settings, a bridge needs to be present that implements the IDataStorage interface that covers platform and OS dependent saving and loading of files. The code:

```
asset.SettingsToXml()
```

will return the serialized Settings.

For the above example the serialized output will look like:

```
<?xml version="1.0" encoding="utf-8"?>
<AssetSettings xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <TestProperty>Hello Default Worldtest</TestProperty>
</AssetSettings>
```

- To use bridge code safely in your asset, a pattern such as displayed below can be used:

```
String fid2 = “MyFile.txt”;
IDataArchive ds = getInterface<IDataArchive>();
if (ds != null)
{
   // Call the appropriate bridge ds.
   //
   ds.Archive(fId2);
}
else
{
   // Default behavior when no bridge is found.
   //
   FileStorage.Remove(fId2);
}
```

Remarks:

-The method getInterface() will look for the specified interface on the asset first. If the interface is not present it checks the RageAssetManager Bridge for the implementation of the interface. If that fails as well, the getInterface() method will return null.

- A bridge is easily implemented in the game Engine code by creating a class that implements at least IBridge (and a number of interfaces), such as:

```
Class Bridge: IBridge, IDataStorage {

}
```

Remarks:

-The IBridge interface is empty and only used for identifying bridge objects.
-The IDataStorage interface is used for saving and loading run-time data.
-The IDataArchive is intended to be used to off-load data (for example prune old run-time after sending it to a server).
-The IDefaultSettings interface is used to load Default Settings that are compiled into the game (so NOT the asset). It also contains a save method that can be used to save a skeleton settings (after creating and initializing it when it is absent) to disk, which allows a programmer to include it in the project and compile it in the game as either a .Net embedded resource or Unity (text) resource.

Platform/engine dependency issues

Please note that the common .Net diagnostic logging with Console.Write or Debug.Write is platform dependent (Mono has some different methods on these classes). Use the ILog interface instead so the game programmer can decide where the logged messages should go.

Remarks:

-The next two steps are optional.

In order to debug in Unity3D not only the assemblies should be dropped in the Unity3D projects Assets folder but also the matching Mono Debug Symbols (these can be converted from the .Net pdb files using the MonoMdbGenerator pdb2mdb.exe).

This can be automated either through a batch file or by including the following afterbuild task at the end of the csproj file:

```
<Target Name="AfterBuild">
   <CallTarget Targets="GenerateMonoSymbols" Condition="
Exists('$(OutputPath)\$(AssemblyName).pdb') " />
  </Target>
```

```
  <Target Name="GenerateMonoSymbols">
    <Message Text="Unity install folder: $(UnityInstallFolder)" Importance="high" />
    <Message Text="$(ProjectName) -&gt; $(TargetPath).mdb" Importance="High" />
    <Exec Command="&quot;$(MonoCLI)&quot; &quot;$(MonoMdbGenerator)&quot;
(AssemblyName).dll" WorkingDirectory="$(MSBuildProjectDirectory)\$(OutputPath)" />
  </Target>
```

Additionally, to determine the location of the pdb2mdb utility, the following lines should be added to the end of the PropertyGroup section of the csproj file:

```
<!-- Look up Unity install folder, and set the ReferencePath for locating managed assembly
references. -->
<UnityInstallFolder>$(registry:HKEY_CURRENT_USER\Software\Unity
Technologies\Installer\Unity@Location
x64)</UnityInstallFolder><ReferencePath>$(UnityInstallFolder)\Editor\Data\</ReferencePath><
MonoFolder>$(UnityInstallFolder)\Editor\Data\MonoBleedingEdge</MonoFolder><MonoMdbG
enerator>$(MonoFolder)\lib\mono\4.5\pdb2mdb.exe</MonoMdbGenerator><MonoCLI>$(Mon
oFolder)\bin\cli.bat</MonoCLI>
```

APPENDIX 3

RAGE

Realising an Applied Gaming Ecosystem

Research and Innovation Action

Grant agreement no.: 644187

WP1 – Creating RAGE client-side assets in TypeScript

RAGE – WP1

FINAL

Project Number	H2020-ICT-2014-1
Due Date	
Actual Date	25-June-2017
Document Author/s	Wim van der Vegt, Wim Westera
Version	1.1
Dissemination level	
Status	Final
Document approved by	

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644187

Document Version Control			
Version	**Date**	**Change Made (and if appropriate reason for change)**	**Initials of Commentator(s) or Author(s)**
0.1	November 2015	Initial version	WimV
0.2	January 2016	Added templates	WimV
1.0	08-May-2017	Updated Document	WimV
1.1	26-June-2017	Updated Document	WimV

Document Change Commentator or Author		
Author Initials	**Name of Author**	**Institution**
WimV	Wim van der Vegt	1.OUNL
WimW	Wim Westera	1.OUNL

Document Quality Control			
Version QA	**Date**	**Comments (and if appropriate reason for change)**	**Initials of QA Person**

Table of Contents

Creating TypeScript Assets

For testing the practicability of the RAGE architecture a Proof-of-Concept asset was developed. The code of this Proof-of-Concept asset is used as a starting point for creating new assets.

The Proof-of-Concept code repository is called '***asset-proof-of-concept-demo_TypeScript'*** and is located on GitHub at https://github.com/rageappliedgame.

The code in this GitHub repository consists of one Visual Studio project with two folders. One is called RageAssetManager and contains both the AssetManager and the base classes for assets. The other folder, RageAsset, contains some simple assets that were used for testing.

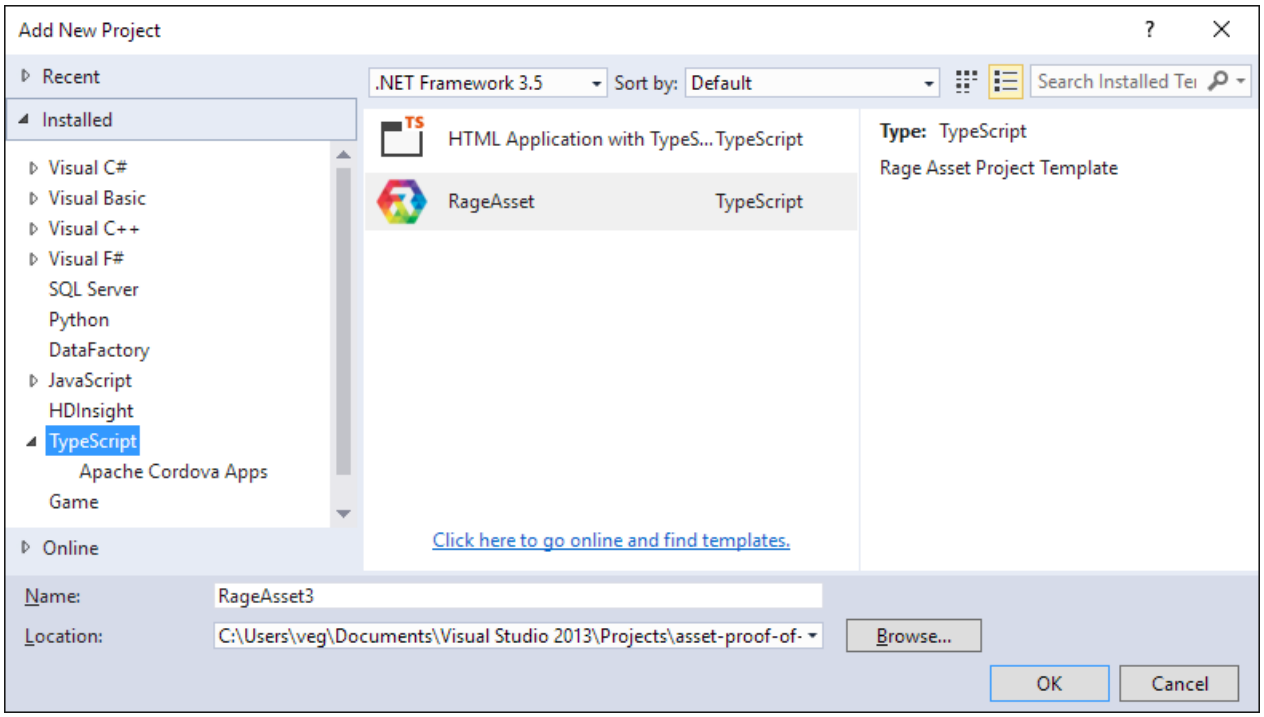
Creating an asset can be done in two ways. One is manually using the proof of concept at GitHub as starting point. The other (more automated) method is to use a Visual Studio template.

Asset Creation using templates.

For this method, a zipped template has to be downloaded from the 'AssetManager-Projects-Templates' repository at GitHub (https://github.com/rageappliedgame). The zip file must be placed in the correct Visual Studio template folder. In a default Visual Studio 2015 installation, the template base folder is a sibling of the project folder (where code is stored). By default, for Visual Studio 2015, the folder to put the template file into is located at '**Visual Studio 2015\Templates\ProjectTemplates\TypeScript**' under **My Documents**.

Remarks:
- The zipped template should not be unzipped.

With the template at the correct location, **File|New Project** will show a new TypeScript project, **RageAsset**.



At the top of the two TypeScript files the path attribute of the /// <reference> directives have to be corrected.

The RageAssetManager project can be retrieved from the '**asset-proof-of-concept-demo_TypeScript**' repository located at GitHub (https://github.com/rageappliedgame), the AssetManager is part of this repository.

-Please note: The AssetManager project can be added to the solution.

-Please note: In the projects settings it might be necessary to check the 'Combine JavaScript output into file' option on the TypeScript Build tab.

Manual Asset Creation

- For creating a new asset in Visual Studio: first create a solution and add the RageAssetManager project to the solution.
- Create a new TypeScript project called MyRageAsset where the Asset being developed will be stored. As default namespace use RageAsset, as module name AssetPackage.
- Add a new class MyAsset for your asset to the RageAsset folder like:

```
/// <reference path="../RageAssetManager/AssetManager.ts"/>
/// <reference path="../RageAssetManager/BaseAsset.ts"/>
/// <reference path="../RageAssetManager/IAsset.ts"/>
///
module AssetPackage {

    // Setup Aliases.
```

```
    import AssetManager = AssetManagerPackage.AssetManager;
    import BaseAsset = AssetPackage.BaseAsset;
    import IAsset = AssetPackage.IAsset;

    /// <summary>
    /// Export the Asset.
    /// </summary>
    export class MyAsset extends BaseAsset {

        /// <summary>
        /// Information describing the protected version.
        /// </summary>
        ///
        /// <remarks>
        /// Commas after the last member and \r\n are not allowed.
        /// </remarks>
        protected versionInfo: string =
        '{ ' +
        '  "Major":"1", ' +
        '  "Minor":"2", ' +
        '  "Build":"3", ' +
        '  "Maturity":"Alpha", ' +
        //'  "Dependencies": [ ' +
        //'        { ' +
        //'           "Class": "OtherAsset", ' +
        //'           "minVersion": "1.0.0", ' +
        //'           "maxVersion": "1.*" ' +
        //'        } ' +
        //'   ] ' +
        '} ';

        /// <summary>
        /// Initializes a new instance of the Asset class. Sets the ClassName
property.
        /// </summary>
        constructor() {
            super();
        }
    }
}
```

Remarks:

-Please note: Unlike the C# implementation the TypeScript version uses natively supported Json instead of xml for storing settings and version info.

- Please note: For retrieving default settings the Bridge object with an IDefaultSettings interface is expected and used.

Compile-time supplied defaults are loaded into the Settings property of the underlying BaseAsset class.

See the sample Bridge.ts implementation which is based on JavaScript's localStorage object present in most modern browsers.

-For saving and loading (run-time) Settings an IDataStorage interface is expected on the bridge object.

- Now the asset can be compiled and can be used in another script.

- When 'compiled', the TypeScript is translated in JavaScript. This resulting code can be used in the browser application.

- For creating the asset (and have it create the RageAssetManager and register itself) add the following lines to your game-engine code:

```
Above the module/namespace block add two references like:

/// <reference path="RageAssetManager/AssetManager.ts"/>

/// <reference path="MyRageAsset/MyAsset.ts"/>

Inside the module/namespace import block and alias the asset:

import MyAsset = AssetPackage.MyAsset;

Finally create an instance with code like:

var asset1 = new MyAsset();
```

- The line below will return a textual report of the assets registered, their versions and (un)solved dependencies:

```
AssetManager.Instance. VersionAndDependenciesReport
```

Creating bridges

A bridge can be added to both the MyAsset and the AssetManager.

- To persist (save/load) the settings a bridge needs to be present that implements the IDataStorage interface that covers platform and OS dependent saving and loading of files. The code:

```
asset.SettingsToJson()
```

will return the serialized Settings.

To use bridge code safely in your asset a pattern such as displayed below can be used:

```
String fid2 = “MyFile.txt”;

var ds: IDataStorage = this.getInterfaceMethod("Load");

if (ds != null)
{
   // Call the appropriate bridge ds.
   //
   ds.Load(fId2);
}
else
{
   // Default behavior when no bridge is found.
   //
}
```

Remarks:

- The method getInterfaceMethod () will look for the specified method on the asset first. If the interface is not present it checks the RageAssetManager Bridge for the implementation of the method. If that fails too the getInterfaceMethod() method will return null.

- Please note: this differs from for example the C# implementation as TypeScript cannot determine the presence of an interface at run-time as the concept does not exist in the resulting JavaScript. Also, it is not possible to obtain a list of all methods of an interface, so a full test on the completeness of the interface is not possible without additional bookkeeping.

- A bridge is easily implemented in the game Engine code by creating a class that implements at least IBridge (and a number of interfaces), such as:

```
/// <reference path="RageAssetManager/IBridge.ts"/>
/// <reference path="RageAssetManager/IDataStorage.ts"/>

module MyNameSpace {

   import IBridge = AssetPackage.IBridge;
   import IDataStorage = AssetPackage.IDataStorage;

   export class Bridge implements IBridge , IDataStorage {

   }

}
```

Remarks:

-The IBridge interface is empty and is only used for identifying bridge objects.
-The IDataStorage interface is used for saving and loading run-time data.
-The IDataArchive is intended to be used to off-load data (for example prune old run-time after sending it to the server).
-The IDefaultSettings interface is used to load Default Settings that are compiled into the game (so NOT the asset). It also contains a save method that can be used to save skeleton settings (after creating and initializing it when it's not present) to disk, which allows a programmer to include it in the project and compile it into the game as a string.

Platform/engine dependency issues

Please note that under Windows 10 the default browser, Edge, does not support debugging TypeScript. So either Internet Explorer or another suitable and supported browser must be used.

ANNEX 4: CODE REVIEW CHECKS FOR CLIENT ASSETS

Compliance with the RAGE client architecture can be (manually) checked via the following checklist.

To be checked at project setup:

- Check the assembly naming ('nnAsset' and 'nnAsset_Portable'), so that the assemblies have different names.
- Namespace in portable projects should match namespace in their counterpart projects.
- Check the default namespace (either the same for both projects or use the default: 'AssetPackage').
- Check for the presence of a 'PORTABLE' symbol define in portable project's Build tab located in the Project settings (this define might be needed when using reflection).

To be checked at project layout:

- Check for the presence of the version info xml in Resources directory (and its naming matches 'nnAsset.VersionAndDependencies.xml`), i.e. the asset has version info.
- Presence of a portable version.
- Has test/demo project.
- Check that there are no dependency on external libraries.

To be checked by source code inspection/search:

- The Asset extends the BaseAsset class.
- Debug output uses the BaseAsset Log method (Both BaseAsset and AssetManager have a Log method) and not the Console or Debug classes.
- Check for hardcoded paths (also in test projects).
- Check for missing test input to projects.
- Usage of a correct Singleton pattern (if needed).
- Access to files via the Bridge.
- Apache License Version 2.0.
- No embedded test codes present in assets (should be in a separate test/demo project).
- Source code documentation (xmldoc).
- Magic values/numbers.

To be checked by compiling portable asset project:

- Presence of an AssemblyInfo.cs file.
- Check for the presence of a *#if !PORTABLE / #endif* block around the *[assembly: Guid]* attribute in the AssemblyInfo.cs file in order to enable compilation of this class.
- No references to System.File.IO (not portable).
- Missing files (i.e. referred to in a project but missing in the repository at GitHub).

To be checked by executing test/demo project:

- Check for hardcoded path in test/demo projects.
- Check for successful test suite completion.

Additional suggestions:

- Clean up unnecessary ‘using’ statements.
- Add test input to the test project as copied content (so they can be accessed without or with a relative path). In Visual Studio, copying of files marked as content can be enabled in the properties of a file.
- Use AssetSettings for configuration data like server address and credentials etc.
- Minimize public methods to API only.

ANNEX 5 Outcomes of the Asset Creation Wizard Usability Study

By Kiavash Bahreini and Enkhbold Nyamsuren, Open Universiteit, Heerlen, The Netherlands
2 June 2017

1. Some main issues reported in the 34 and 42 open responses

a. Apart from this issue that entering metadata is not fun for asset developers, they reported that they would use this metadata editor when they have a new asset to introduce or if major changes occurs.
b. Some fields are tricky. Asset developers might not have a logo for their own asset. Game development environment and Target Platform should be either a choice on it's own or the field should be a set of checkboxes.
c. The categories in the ACM classification version 2012 are either overly generic ("games") or widely inapplicable for the types of assets. Learning goals are also not especially useful.
d. The most important tags may be the user-defined keywords, but those are unlikely to be consistent between asset creators.
e. It is expected to lose your data if you do not fill in the 'mandatory' fields.
f. There are 8 screens of information before being able to actually save progress. This qualifies as moderately cumbersome.
g. The purpose of some fields, such as technical description, full description, and short description should be clearly explained in more details. Thus, the users may understand if they need to provide more or less information. Some tips, such as "this will be used in the assets gallery" and "this will be displayed on the dedicated page of the asset", etc. are proposed.
h. The asset developers mentioned that they could fill in most of the fields quickly, however they had to skip some fields that were beyond their knowledge or not matching the asset. Some fields seemed duplicates like the field in Step 6: detailed description and some fields in Step 2: technical description. They mentioned that 8 pages take some time to scan and read. The others mentioned that the first section was the hardest section, but the cloning function could help them to start with an asset completed in percent of about 60%. Furthermore they mentioned that the 4 description fields (short, detailed, technical, etc.) took most of the time from them.
i. Mandatory fields were clearly visible.
j. When more than one asset developers working on the same asset, they could potentially use very different terms to explain the asset functionalities. Therefore, they may have different expectation on which information is expected from them.
k. Asset developers did not clearly know at every input what rule they had to stick. There was no indication of max size for all the input fields. This raised a problem with their text for detailed description [optional] field at step 6, where the input text was not saved after submission, without any indications about this limit. The information related to the length and date format is missing. Most answer lengths are either one line or four lines. In the four lines cases, there is no "expected length" indicator beyond a phrase count; and no formatting support to help structure potentially long answers to make them more readable when displayed.
l. The asset developers reported that, except two cases, the fill in was not fully eased by given answers using drop-down menus and checkboxes. Some fields did not match the expectations and most of the applied computing concepts are of little relevance to

pedagogical computer game assets. They reported that the excessively long drop down menus made searching hard. Plus the metadata editor did not include a general games category.

m. In case of a problem and instruction by an error message on how to solve the problem, the asset developers have mentioned the following important issues:
    i. One got 98% completeness in the about section. Not sure why it is not 100%. One could not see any error messages. One has mentioned that there was no indication of max size for all the input fields. This raised a problem with our text for the ‘detailed description [optional]’ field at Step 6, where the input text was not saved after submission, without any indications about this limit. When one asset developer tried to submit an asset, he encountered with an error message that the asset could not be saved due to either a lost connection to the server or that could some errors in a given answer. The asset developer had to review everything to find in the end that the URL of documentation had a space character and no error message was given. One received no error message when losing all unsaved data due to accidentally navigating away from page; and no way to retrieve unsaved data. When searching for an asset and no matches are found, "Cannot retrieve the list of software assets" was displayed instead of "no results match this query".
n. In reply to the question whether the purpose and utility of the metadata editor was clear to you, the asset developers mentioned the following issues:
    i. One mentioned that it is not clear to him and the other mentioned that he does not think the overall purpose of the asset creation wizard will be clear to the developers outside of the RAGE project. What the developers should do with the created metadata? The other one mentioned that the main goal of metadata is to be searchable. Search currently only looks at full matches starting from the beginning of the asset title. The other did not understand whether the metadata editor is just an interface for the final "app store", which will be later developed, or it is in fact the app store.
o. Removing an uploaded file does not seem to work.
p. Some assets contain no setup files, as they are only source code; so the ‘setup files field’ can’t be mandatory. This will allow the asset developer to submit the asset.
q. The final report about the percentage of the filled in asset has no meaning to the asset developers at the moment.

2. Concluding observation based on all instruments and comments

a. The opinion of the participants in the usability study show that they are moderately positive towards using the metadata editor.
b. Participants provided 34 (mean=3.4, standard error=.56) and 42 (mean=4.2, standard error=.7) comments to their responses in the SUS and the FUS questionnaires, respectively. While overall usability scores are positive, the number of comments indicates that there may be some specific issues in the Asset creation wizard that should be further resolved. Finally, more comments in the FUS questionnaire indicate that the FUS questionnaire was able to capture the issues that are specific to online forms: as we expected.
c. SUS results (mean=72.83, standard deviation= 16.31, standard error=4.21) indicate that the overall usability evaluation is positive. All the participants except one positively evaluated the overall usability of the Asset creation wizard. The mean scores for all questions except one are positive. The question with the negative mean score

(mean=1.73, standard error=.27) is concerned with the frequency of using the Asset creation wizard. The negative score is expected because the Asset creation wizard should not be frequently used.

d. FUS results (mean=65.74, standard deviation=15.98, standard error=4.13) indicate that the overall usability evaluation of the Asset creation wizard is positive with some rooms for improvement. Two participants negatively evaluated the overall usability of the Asset creation wizard. The correlation between the SUS and the FUS overall scores is significantly high ($r(13) = .67$, $p = .006$), which indicates that the FUS score is consistent with the benchmark score of the SUS. The overall FUS score is lower than the overall SUS score. The lower score might indicate that the FUS questionnaire is able to identify issues specific to online forms. To further investigate this matter, we look at the scores of the individual FUS questions. The mean scores for nine questions are positive. One question has a negative mean score (mean=1.87, standard error=.33). The responses to this question indicate that the Asset creation wizard did not have sufficient feedback to users for resolving unexpected problems. Another question with the lowest mean positive score (mean=2.33, standard error=.29) is close to the neutral score of 2. The responses to this question indicate that the participants had some difficulties understanding what information was expected to enter into the Asset creation wizard.
e. UMUX results (mean=72.5, standard deviation=16.67, standard error=4.3) indicate that the distribution of the overall scores is positive with only one overall score being negative. The correlation of the UMUX scores with the SUS scores is significantly high ($r(13) = .88$, $p < .001$). The correlation of the UMUX scores with the FUS scores is significantly high as well ($r(13) = .74$, $p = .002$). The results indicate that the evaluations of the three usability components are positive. The Asset creation wizard is fit for its purpose for managing metadata. The participants also reported positive efficiency indicating that metadata management was fast and did not require substantial effort. Finally, the participants reported positive satisfaction towards using the Asset creation wizard.

3. Critical actions to improve the quality

a. The metadata editor does not support ENTERS KEY for new lines.
b. The metadata editor generally works as expected, but losing all data if you navigate away from the page accidentally before "submitting" a half-complete asset description is highly off-putting.
c. There seems to be some sort of redundancy regarding some fields that require further improvement.
d. The metadata editor should support saving data after every section and not only at the end of the last section (section 8).
e. Some tips, such as "this will be used in the assets gallery" and "this will be displayed on the dedicated page of the asset", etc. are proposed to use in the following fields: technical description, full description, and short description. Using these, the asset developer may understand if they need to provide more or less information.
f. It is not really necessary to study a dedicated material before using the metadata editor. Maybe some "business manual" or something similar that could help the asset developer to better emphasize the ideas of the asset and to gain a more downloads.
g. The length of the metadata editor could have been shorter. It should also allow the developer to save draft data. Some fields show a little redundancy. The redundancy might me decreased if every field will be better explained.
h. Duplication in entering some metadata and text in some fields should be removed.
i. All the links to the asset and to the sources should be provided in a similar step.

j. Some tips are required to complete some fields.
k. Importing external files, such as a ‘readme.txt’ file should be smoothly improved.
l. The information related to the max size, length, and date format are expected to develop in the next version of the metadata editor.
m. The current error message function of the metadata editor is not fully operative. The developers of the metadata editor should further improve it in future.
n. The maximum size of some fields should be clearly stated.
o. The asset developers might easily loose the entered data if any unwanted error occurs during the data entry to the metadata editor and in the worst-case scenario at the end of step 8.
p. The overall utility of the metadata editor should be further explained to asset developers.
q. The current search functionality of the metadata editor will only find out the asset based on the name of the asset developer(s), but in some cases the asset should be found out by its owner’s name or by other input fields.
r. Removing an uploaded file, load/save functionality, sharing functionality, and editing the asset should be further improved or developed.
s. Integration with git version control systems to download and zip a commit would be nice.
t. In the final step, one received a message that he missed some mandatory fields. It's not clear in which steps these are located. An indicator on the tabs (or the submission percentage list) might help.